Вильямс
OSBORNE
McGraw Hill

Обновленный и дополненный классический справочник Шилдта по C++

Полный справочник

по C++

Четвертое издание

Содержит описание международного стандарта C++, в том числе ключевых слов, синтаксиса и библиотек

Рассмотрены новейшие свойства языка – в частности, перегрузка, наследование, виртуальные функции, пространства имен, шаблоны, стандартная библиотека шаблонов STL и механизм RTTI

Примеры программ совместимы со всеми компиляторами языка C++, включая Visual C++

Бесплатный код в Internet

Герберт Шилдт

Автор наиболее популярных книг по программированию. Количество проданных экземпляров превышает 3 миллиона!

Полный справочник по C++

4-е издание

C++: The Complete Reference

Fourth Edition

Herbert Schildt

McGraw-Hill/Osborne

New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

Полный справочник по C++

4-е издание

Герберт Шилдт

Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2006

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. физ.-мат. наук *Д.А. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт.

Ш57 Полный справочник по C++, 4-е издание. . Пер. с англ. — М. : Издательский дом "Вильямс", 2006. — 800 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0489-7 (рус.)

В четвертом издании этой книги полностью описаны и проиллюстрированы все ключевые слова, функции, классы и свойства языка C++, соответствующие стандарту ANSI/ISO. Информацию, изложенную в книге, можно использовать во всех современных средах программирования. Освещены все аспекты языка C++, включая его основу — язык C. Справочник состоит из пяти частей: 1) подмножество C; 2) язык C++; 3) библиотека стандартных функций; 4) библиотека стандартных классов; 5) приложения на языке C++.

Книгу предназначена для широкого круга программистов.

ББК 32.973.26-018.2.75

ISBN 5-8459-0489-7 (рус.)
ISBN 0-07-222680-3 (англ.)

Оглавление

Об авторе 24

Введение 25

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА C++: ПОДМНОЖЕСТВО C 27

Глава 1. Обзор языка C 29

Глава 2. Выражения 39

Глава 3. Операторы 73

Глава 4. Массивы и строки 99

Глава 5. Указатели 117

Глава 6. Функции 135

Глава 7. Структуры, объединения, перечисления и оператор typedef 153

Глава 8. Ввод-вывод на консоль 173

Глава 9. Файловый ввод-вывод 191

Глава 10. Препроцессор и комментарии 211

ЧАСТЬ II. ЯЗЫК C++ 223

Глава 11. Обзор языка C++ 225

Глава 12. Классы и объекты 251

Глава 13. Массивы, указатели, ссылки и операторы динамического распределения памяти 279

Глава 14. Перегрузка функций, конструкторы копирования и аргументы по умолчанию 303

Глава 15. Перегрузка операторов 321

Глава 16. Наследование 347

Глава 17. Виртуальные функции и полиморфизм 367

Глава 18. Шаблоны 379

Глава 19. Обработка исключительных ситуаций 399

Глава 20. Основы системы ввода-вывода 415

Глава 21. Файловая система 437

Глава 22. Динамическая идентификация типа и операторы приведения 457

Глава 23. Пространства имен, преобразования функций и другие новшества 475
Глава 24. Введение в стандартную библиотеку шаблонов 503
ЧАСТЬ III. БИБЛИОТЕКА СТАНДАРТНЫХ ФУНКЦИЙ 553
Глава 25. Функции ввода-вывода языка C 555
Глава 26. Строковые и символьные функции 573
Глава 27. Математические функции 583
Глава 28. Функции времени, даты и локализации 591
Глава 29. Функции динамического распределения памяти 597
Глава 30. Служебные функции 601
Глава 31. Функции обработки расширенных символов 613
ЧАСТЬ IV. БИБЛИОТЕКА СТАНДАРТНЫХ КЛАССОВ 621
Глава 32. Стандартные классы ввода-вывода 623
Глава 33. Стандартные контейнерные классы 641
Глава 34. Стандартные алгоритмы 661
Глава 35. Стандартные итераторы, распределители памяти и функторы 677
Глава 36. Класс string 693
Глава 37. Числовые классы 703
Глава 38. Обработка исключительных ситуаций и прочие классы 723
ЧАСТЬ V. ПРИЛОЖЕНИЯ НА ЯЗЫКЕ C++ 729
Глава 39. Интеграция новых классов: пользовательский класс для работы со строками 731
Глава 40. Синтаксический анализ выражений 751
Приложение А. Расширение языка C++ для платформы .NET 779
Приложение Б. Язык C++ и робототехника 783
Предметный указатель 787

Содержание

Об авторе 24

Введение 25

Изменения, внесенные в четвертое издание 25
Содержание книги 25
Рекомендуемая литература 26

ЧАСТЬ I. ОСНОВЫ ЯЗЫКА C++: ПОДМНОЖЕСТВО C 27

Глава 1. Обзор языка C 29

Происхождение и история языка C 30
Сравнение стандартов C89 и C99 30
C — язык среднего уровня 31
C — структурированный язык 32
C — язык для программистов 34
Структура программы на языке C 35
Библиотека и связывание 36
Раздельная компиляция 37
Расширения файлов .C и .CPP 37

Глава 2. Выражения 39

Пять основных типов данных 40
Модификация основных типов 41
Идентификаторы 42
Переменные 43
Где объявляются переменные 43
Локальные переменные 43
Формальные параметры 45
Глобальные переменные 46
Квалификаторы const и volatile 47
Квалификатор const 47
Квалификатор volatile 48
Спецификаторы хранения 49
Спецификатор extern 49
Статические переменные 51
Локальные статические переменные 51
Глобальные статические переменные 52
Спецификатор register 53
Инициализация переменных 54
Константы 54
Шестнадцатеричные и восьмеричные константы 55
Строковые константы 55
Управляющие символьные константы 56
Операторы 56
Оператор присваивания 57
Преобразования типов в операторе присваивания 57
Множественные присваивания 58
Арифметические операторы 58

Инкрементация и декрементация 59
Операторы сравнения и логические операторы 60
Побитовые операторы 62
Тернарный оператор 65
Операторы взятия адреса и разыменования 66
Статический оператор sizeof 67
Оператор последовательного вычисления 68
Оператор доступа к члену структуры и ссылки на член структуры 68
Операторы "[]" и "()" 69
Приоритеты операторов 69
Выражения 70
Порядок вычислений 70
Преобразование типов в выражениях 70
Приведение типов 71
Пробелы и круглые скобки 72
Составные операторы присваивания 72
Глава 3. Операторы 73
Истинные и ложные значения в языках C и C++ 74
Условные операторы 75
Оператор if 75
Вложенные операторы if 76
Цепочка операторов if-then-else 77
Тернарная альтернатива 78
Условное выражение 80
Оператор switch 81
Вложенные операторы switch 83
Операторы цикла 84
Цикл for 84
Варианты цикла for 85
Бесконечный цикл 88
Пустой цикл for 89
Цикл while 89
Цикл do-while 91
Объявление переменных в условных операторах и циклах 92
Операторы перехода 93
Оператор return 93
Оператор goto 93
Оператор break 94
Функция exit 95
Оператор continue 96
Операторы-выражения 97
Блок 97
Глава 4. Массивы и строки 99
Одномерные массивы 100
Создание указателя на массив 101
Передача одномерного массива в функцию 101
Строки, завершающиеся нулевым байтом 103
Двухмерные массивы 104
Массивы строк 107
Многомерные массивы 108
Индексация указателей 109

Инициализация массива 111
Инициализация безразмерного массива 112
Игра в крестики-нолики 113

Глава 5. Указатели 117

Что такое указатель 118
Указатели 118
Операторы для работы с указателями 119
Выражения, содержащие указатели 120
Присваивание указателей 120
Адресная арифметика 120
Сравнение указателей 121
Указатели и массивы 123
Массивы указателей 124
Косвенная адресация 125
Инициализация указателей 126
Указатели на функции 127
Функции динамического распределения памяти 130
Проблемы, возникающие при работе с указателями 131

Глава 6. Функции 135

Общий вид функции 136
Область видимости функции 136
Аргументы функции 137
Передача параметров по значению и по ссылке 137
Передача параметров по ссылке 138
Передача массивов в качестве параметров 139
Аргументы функции main(): argc и argv 141
Оператор return 143
Возврат управления из функции 143
Возвращаемые значения 145
Возврат указателей 146
Функции типа void 147
Зачем нужен оператор return в функции main() 148
Рекурсия 148
Прототипы функций 149
Прототипы стандартных библиотечных функций 151
Определение списка параметров переменной длины 151
Объявление параметров функции в классическом и современном стиле 152

Глава 7. Структуры, объединения, перечисления и оператор typedef 153

Структуры 154
Доступ к членам структуры 156
Присваивание структур 157
Массивы структур 157
Передача структур функциям 158
Передача членов структур 158
Передача целых структур 158
Указатели на структуры 160
Объявление указателей на структуры 160
Использование указателей на структуры 160
Массивы и структуры внутри структур 162
Битовые поля 163
Объединения 165

Перечисления 167
Применение оператора sizeof для обеспечения машинонезависимости 169
Оператор typedef 170

Глава 8. Ввод-вывод на консоль 173

Важное замечание прикладного характера 174
Чтение и запись символов 175
Проблемы, связанные с функцией getchar() 176
Альтернативы функции getchar() 176
Чтение и запись строк 177
Форматированный ввод-вывод на консоль 179
Функция printf() 179
Вывод символов 180
Вывод чисел 180
Вывод адресов 181
Спецификатор %n 181
Модификаторы формата 182
Модификатор минимальной ширины поля 182
Модификатор точности 183
Выравнивание вывода 184
Обработка данных других типов 184
Модификаторы * и # 185
Функция scanf() 185
Спецификаторы формата 186
Ввод чисел 186
Ввод целых чисел без знака 187
Ввод отдельных символов 187
Ввод строк 187
Ввод адреса 188
Спецификатор %n 188
Использование набора сканируемых символов 188
Пропуск нежелательных разделителей 189
Символы, не являющиеся разделителями 189
Функции scanf() следует передавать адреса 189
Модификаторы формата 189
Подавление ввода 190

Глава 9. Файловый ввод-вывод 191

Файловые системы языков C и C++ 192
Потоки и файлы 192
Потоки 192
Текстовые потоки 193
Бинарные потоки 193
Файлы 193
Основы файловой системы 194
Указатель файла 194
Открытие файла 195
Закрытие файла 196
Запись символа 196
Чтение символа 197
Применение функций fopen(), getc(), putc() и fclose() 197
Применение функции feof() 198
Работа со строками: функции fputs() и fgets() 200

Функция rewind() 200
Функция ferror() 201
Удаление файла 203
Очистка потока 203
Функции fread() и fwrite() 204
Применение функций fread() и fwrite() 204
Функция fseek() и файлы с произвольным доступом 205
Функции fprintf() и fscanf() 207
Стандартные потоки 208
Связь с консольным вводом-выводом 209
Применение функции freopen() для перенаправления стандартных потоков 209

Глава 10. Препроцессор и комментарии 211

Препроцессор 212
Директива #define 212
Определение функций в виде макросов 214
Директива #error 214
Директива #include 215
Директивы условной компиляции 215
Директивы #if, #else, #elif и #endif 215
Директивы #ifdef и ifndef 217
Директива #undef 218
Оператор defined 219
Директива #line 219
Директива #pragma 220
Операторы препроцессора # и ## 220
Имена предопределенных макросов 221
Комментарии 221
Однострочные комментарии 222

ЧАСТЬ II. ЯЗЫК C++ 223

Глава 11. Обзор языка C++ 225

Истоки языка C++ 226
Что такое объектно-ориентированное программирование 227
Инкапсуляция 228
Полиморфизм 228
Наследование 229
Некоторые основные принципы языка C++ 229
Пример программы на языке C++ 229
Операторы ввода-вывода 232
Объявление локальных переменных 232
Правило “int по умолчанию” 233
Тип данных bool 234
Старый и новый стиль языка C++ 234
Новый стиль заголовков 235
Пространства имен 236
Работа со старым компилятором 237
Введение в классы 237
Перегрузка функций 240
Перегрузка операторов 243
Наследование 243
Конструкторы и деструкторы 247

Ключевые слова языка C++ 250
Структура программы на языке C++ 250
Глава 12. Классы и объекты 251
Классы 252
Связь между структурами и классами 254
Связь между объединениями и классами 256
Безымянные объединения 257
Дружественные функции 258
Дружественные классы 261
Подставляемые функции 262
Определение подставляемых функций внутри класса 264
Конструкторы с параметрами 265
Конструкторы с одним параметром: особый случай 267
Статические члены класса 267
Статические переменные-члены 267
Статические функции-члены 271
Вызов конструкторов и деструкторов 272
Оператор разрешения области видимости 274
Вложенные классы 274
Локальные классы 274
Передача объектов функциям 275
Возвращение объектов 277
Присваивание объектов 278
Глава 13. Массивы, указатели, ссылки и операторы динамического распределения памяти 279
Массивы объектов 280
Инициализированные и неинициализированные массивы 282
Указатели на объекты 282
Проверка типа указателей 284
Указатель this 284
Указатели на производные типы 285
Указатели на члены класса 287
Ссылки 289
Передача параметров с помощью ссылок 290
Передача ссылок на объекты 292
Возврат ссылок 293
Независимые ссылки 294
Ссылки на производные типы 294
Ограничения на ссылки 295
Стиль 295
Операторы динамического распределения памяти 295
Инициализация выделяемой памяти 297
Выделение памяти для массивов 297
Выделение памяти для объектов 298
Альтернатива nothrow 302
Буферизованный оператор new 302
Глава 14. Перегрузка функций, конструкторы копирования и аргументы по умолчанию 303
Перегрузка функций 304
Перегрузка конструкторов 305

Перегрузка конструктора для достижения гибкости 305
Создание инициализированных и неинициализированных объектов 307
Конструктор копирования 308
Определение адреса перегруженой функции 311
Анахронизм overload 312
Аргументы функции по умолчанию 312
Аргументы по умолчанию и перегрузка 316
Правильное применение аргументов по умолчанию 317
Перегрузка функций и неоднозначность 317

Глава 15. Перегрузка операторов 321

Создание операторной функции-члена 322
Создание префиксной и постфиксной форм операторов инкрементации и декрементации 326
Перегрузка сокращенных операторов присваивания 327
Ограничения на перегруженные операторы 327
Перегрузка операторов с помощью дружественных функций 327
Применение дружественных функций для перегрузки операторов "++" и "——" 329
Дружественные операторные функции повышают гибкость 331
Перегрузка операторов new и delete 332
Перегрузка операторов new и delete для массивов 336
Перегрузка операторов new и delete, не генерирующих исключительной ситуации 338
Перегрузка некоторых специальных операторов 339
Перегрузка оператора "[]" 339
Перегрузка оператора "()" 342
Перегрузка оператора "->" 343
Перегрузка оператора "," 344

Глава 16. Наследование 347

Управление доступом к членам базового класса 348
Наследование и защищенные члены 349
Защищенное наследование 352
Множественное наследование 353
Конструкторы, деструкторы и наследование 354
Когда вызываются конструкторы и деструкторы 354
Передача параметров конструктору базового класса 357
Предоставление доступа 360
Виртуальные базовые классы 362

Глава 17. Виртуальные функции и полиморфизм 367

Виртуальные функции 368
Вызов виртуальной функции с помощью ссылки на объект базового класса 370
Атрибут virtual наследуется 371
Виртуальные функции являются иерархическими 372
Чисто виртуальные функции 374
Абстрактные классы 376
Применение виртуальных функций 376
Сравнение раннего и позднего связывания 378

Глава 18. Шаблоны 379

Обобщенные функции 380
Функция с двумя обобщенными типами 382
Явная перегрузка обобщенной функции 382

Перегрузка шаблонной функции 384
Использование стандартных параметров шаблонных функций 385
Ограничения на обобщенные функции 385
Применение обобщенных функций 386
Обобщенная сортировка 386
Уплотнение массива 388
Обобщенные классы 389
Пример использования двух обобщенных типов данных 391
Применение шаблонных классов: обобщенный массив 392
Применение стандартных типов в обобщенных классах 393
Применение аргументов по умолчанию в шаблонных классах 395
Явные специализации классов 396
Ключевые слова typename и export 397
Мощь шаблонов 398

Глава 19. Обработка исключительных ситуаций 399

Основы обработки исключительных ситуаций 400
Перехват классов исключительных ситуаций 404
Применение нескольких операторов catch 405
Обработка производных исключительных ситуаций 406
Тонкости обработки исключительных ситуаций 407
Перехват всех исключительных ситуаций 407
Ограничения исключительных ситуаций 409
Повторное генерирование исключительной ситуации 410
Функции terminate() и unexpected() 411
Обработчики, связанные с функциями terminate() и unexpected() 412
Функция uncaught_exception() 413
Классы exception и bad_exception 413
Применение обработки исключительных ситуаций 413

Глава 20. Основы системы ввода-вывода 415

Сравнение старой и новой систем ввода-вывода 416
Потоки 416
Классы потоков в языке C++ 417
Встроенные потоки в языке C++ 418
Форматированный ввод-вывод 418
Форматирование с помощью членов класса ios 418
Установка флагов формата 419
Сброс флагов формата 420
Перегруженная форма функции setf() 421
Проверка флагов форматирования 422
Установка всех флагов 423
Применение функций width(), precision() и fill() 423
Применение манипуляторов формата 425
Перегрузка операторов “<<” и “>>” 427
Создание собственных функций вставки 427
Создание собственных функций извлечения 432
Создание собственных манипуляторов 434

Глава 21. Файловая система 437

Заголовок <fstream> и классы файлов 438
Открытие и закрытие файла 438
Чтение и запись текстовых файлов 440
Бесформатный и бинарный ввод-вывод 442

Сравнение символов и байтов 442
Функции put() и get() 442
Функции read() и write() 444
Дополнительные функции get() 446
Функция getline() 446
Распознавание конца файла 447
Функция ignore() 449
Функции peek() и putback() 450
Функция flush() 450
Произвольный доступ 450
Определение текущей позиции 453
Статус ввода-вывода 453
Настройка ввода-вывода в файлы 455

Глава 22. Динамическая идентификация типа и операторы приведения 457

Динамическая идентификация типа (RTTI) 458
Применение динамической идентификации типа 462
Применение оператора typeid к шаблонным классам 464
Операторы приведения типов 465
Оператор dynamic_cast 465
Замена оператора typeid оператором dynamic_cast 468
Применение оператора dynamic_cast к шаблонным классам 470
Оператор const_cast 471
Оператор static_cast 473
Оператор reinterpret_cast 473

Глава 23. Пространства имен, преобразования функций и другие новшества 475

Пространства имен 476
Основы пространств имен 476
Директива using 479
Неименованные пространства имен 480
Некоторые особенности пространств имен 481
Пространство имен std 483
Создание функций преобразования 484
Функции-члены с атрибутами const и mutable 487
Функции-члены с атрибутом volatile 489
Явные конструкторы 489
Инициализация членов класса 490
Применение ключевого слова asm 494
Спецификации связей 495
Буферизованный ввод-вывод 495
Классы буферизованного вывода 496
Создание буферизованного потока вывода 496
Применение буферизованного ввода 497
Буферизованный ввод-вывод 499
Применение динамических массивов 499
Применение бинарных операций ввода-вывода к буферизованным потокам 500
Отличия между языками C и C++ 501

Глава 24. Введение в стандартную библиотеку шаблонов 503

Обзор библиотеки STL 504
Контейнеры 504
Алгоритмы 505

Итераторы 505
Другие элементы библиотеки STL 505
Контейнерные классы 506
Общие принципы функционирования 507
Векторы 508
Доступ к элементам вектора с помощью итератора 511
Вставка и удаление элементов вектора 513
Вектор, содержащий объекты класса 514
Списки 516
Функция end() 519
Сравнение функций push_front() и push_back() 520
Сортировка списка 521
Вставка одного списка в другой 521
Список, содержащий объекты класса 523
Ассоциативные контейнеры 524
Ассоциативный массив, содержащий объекты 528
Алгоритмы 529
Подсчет 531
Удаление и замена элементов 533
Изменение порядка следования элементов последовательности 535
Преобразование последовательности 535
Применение функторов 537
Унарные и бинарные функторы 537
Применение встроенных функторов 537
Создание функтора 539
Применение редакторов связей 541
Класс string 542
Некоторые функции — члены класса string 546
Основные манипуляторы 546
Поиск символа в строке 548
Сравнение строк 550
Создание С-строки 550
Строки как контейнеры 550
Запись строки в другой контейнер 551
Заключительные замечания о библиотеке STL 552

ЧАСТЬ III. БИБЛИОТЕКА СТАНДАРТНЫХ ФУНКЦИЙ 553

Глава 25. Функции ввода-вывода языка С 555

Функция clearerr 556
Функция fclose 556
Функция feof 557
Функция ferror 557
Функция fflush 557
Функция fgetc 557
Функция fgetpos 558
Функция fgets 558
Функция fopen 558
Функция fprintf 559
Функция fputc 560
Функция fputs 560
Функция fread 560
Функция freopen 560

Функция fscanf 561
Функция fseek 561
Функция fsetpos 562
Функция ftell 562
Функция fwrite 562
Функция getc 562
Функция getchar 563
Функция gets 563
Функция perror 563
Функция printf 564
Функция putc 566
Функция putchar 566
Функция puts 566
Функция remove 567
Функция rename 567
Функция rewind 567
Функция scanf 567
Функция setbuf 570
Функция setvbuf 570
Функция sprintf 570
Функция sscanf 571
Функция tmpfile 571
Функция tmpnam 571
Функция ungetc 571
Функции vprintf, vfprintf и vsprintf 572

Глава 26. Строковые и символьные функции 573

Функция isalnum 574
Функция isalpha 574
Функция iscntrl 574
Функция isdigit 575
Функция isgraph 575
Функция islower 575
Функция isprint 575
Функция ispunct 575
Функция isspace 575
Функция isupper 576
Функция isxdigit 576
Функция memchr 576
Функция memcmp 576
Функция memcpy 576
Функция memmove 577
Функция memset 577
Функция strcat 577
Функция strchr 577
Функция strcmp 577
Функция strcoll 578
Функция strcpy 578
Функция strcspn 578
Функция strerror 578
Функция strlen 579
Функция strncat 579
Функция strncmp 579
Функция strncpy 579

Функция strpbrk 580
Функция strrchr 580
Функция strspn 580
Функция strstr 580
Функция strtok 580
Функция strxfrm 581
Функция tolower 581
Функция toupper 581

Глава 27. Математические функции 583

Функция acos 584
Функция asin 584
Функция atan 585
Функция atan2 585
Функция ceil 585
Функция cos 585
Функция cosh 585
Функция exp 586
Функция fabs 586
Функция floor 586
Функция fmod 586
Функция frexp 586
Функция ldexp 587
Функция log 587
Функция log10 587
Функция modf 587
Функция pow 587
Функция sin 588
Функция sinh 588
Функция sqrt 588
Функция tan 588
Функция tanh 589

Глава 28. Функции времени, даты и локализации 591

Функция asctime 592
Функция clock 592
Функция ctime 593
Функция difftime 593
Функция gmtime 593
Функция localeconv 593
Функция localetime 594
Функция mktime 595
Функция setlocale 595
Функция strftime 595
Функция time 596

Глава 29. Функции динамического распределения памяти 597

Функция calloc 598
Функция free 598
Функция malloc 598
Функция realloc 599

Глава 30. Служебные функции 601

Функция abort 602

Функция abs 602
Макрос assert 602
Функция atexit 603
Функция atof 603
Функция atoi 603
Функция atol 604
Функция bsearch 604
Функция div 604
Функция exit 605
Функция getenv 605
Функция labs 605
Функция ldiv 605
Функция longjmp 606
Функция mblen 606
Функция mbstowcs 606
Функция mbtowc 607
Функция qsort 607
Функция raise 607
Функция rand 608
Функция setjmp 608
Функция signal 608
Функция srand 609
Функция strtod 609
Функция strtol 609
Функция strtoul 610
Функция system 610
Функции va_arg, va_start и va_end 611
Функция wcstombs 611
Функция wctomb 611

Глава 31. Функции обработки расширенных символов 613

Функции классификации расширенных символов 614
Функции ввода-вывода расширенных символов 616
Функции обработки строк, состоящих из расширенных символов 617
Функции преобразования строк, состоящих из расширенных символов 618
Функции обработки массивов расширенных символов 618
Функции преобразования многобайтовых и расширенных символов 619

ЧАСТЬ IV. БИБЛИОТЕКА СТАНДАРТНЫХ КЛАССОВ 621

Глава 32. Стандартные классы ввода-вывода 623

Классы ввода-вывода 624
Заголовки ввода-вывода 625
Флаги форматирования и манипуляторы ввода-вывода 626
Некоторые типы данных 627
Типы streamsize и streamoff 627
Типы streampos и wstreampos 627
Типы pos_type и off_type 628
Тип openmode 628
Тип iostate 628
Тип seekdir 628
Класс failure 628
Перегрузка операторов “<<” и “>>” 629

Универсальные функции ввода-вывода 629
Функция bad 629
Функция clear 629
Функция cof 629
Функция exceptions 630
Функция fail 630
Функция fill 630
Функция flags 630
Функция flush 630
Функции fstream, ifstream и ofstream 631
Функция gcount 631
Функция get 631
Функция getline 632
Функция good 633
Функция ignore 633
Функция open 633
Функция peek 634
Функция precision 634
Функция put 634
Функция putback 634
Функция rdstate 635
Функция read 635
Функция readsome 635
Функции seekg и seekp 635
Функция setf 636
Функция setstate 637
Функция str 637
Функции stringstream, istringstream и ostringstream 637
Функция sync_with_stdio 638
Функции tellg и tellp 638
Функция unsetf 639
Функция width 639
Функция write 639

Глава 33. Стандартные контейнерные классы 641

Контейнерные классы 642
Класс bitset 643
Класс deque 645
Класс list 646
Класс map 648
Класс multimap 650
Класс multiset 652
Класс queue 654
Класс priority_queue 655
Класс set 655
Класс stack 657
Класс vector 658

Глава 34. Стандартные алгоритмы 661

Алгоритм adjacent_find 662
Алгоритм binary_search 662
Алгоритм copy 662
Алгоритм copy_backward 663
Алгоритм count 663

Алгоритм count_if 663
Алгоритм equal 663
Алгоритм equal_range 663
Алгоритмы fill и fill_n 664
Алгоритм find 664
Алгоритм find_end 664
Алгоритм find_first_of 664
Алгоритм find_if 665
Алгоритм for_each 665
Алгоритмы generate и generate_n 665
Алгоритм includes 665
Алгоритм inplace_merge 665
Алгоритм iter_swap 666
Алгоритм lexicographical_compare 666
Алгоритм lower_bound 666
Алгоритм make_heap 666
Алгоритм max 667
Алгоритм max_element 667
Алгоритм merge 667
Алгоритм min 667
Алгоритм min_element 668
Алгоритм mismatch 668
Алгоритм next_permutation 668
Алгоритм nth_element 668
Алгоритм partial_sort 669
Алгоритм partial_sort_copy 669
Алгоритм partition 669
Алгоритм pop_heap 669
Алгоритм prev_permutation 670
Алгоритм push_heap 670
Алгоритм random_shuffle 670
Алгоритмы remove, remove_if, remove_copy и remove_copy_if 670
Алгоритмы replace, replace_copy, replace_if и replace_copy_if 671
Алгоритмы reverse и reverse_copy 672
Алгоритмы rotate и rotate_copy 672
Алгоритм search 672
Алгоритм search_n 672
Алгоритм set_difference 673
Алгоритм set_intersection 673
Алгоритм set_symmetric_difference 673
Алгоритм set_union 674
Алгоритм sort 674
Алгоритм sort_heap 674
Алгоритм stable_partition 675
Алгоритм stable_sort 675
Алгоритм swap 675
Алгоритм swap_ranges 675
Алгоритм transform 675
Алгоритм unique и unique_copy 676
Алгоритм upper_bound 676

Глава 35. Стандартные итераторы, распределители памяти и функторы 677

Итераторы 678
Основные типы итераторов 678

Классы низкоуровневых итераторов 679
Класс iterator 679
Класс iterator_traits 679
Встроенные итераторы 679
Класс insert_iterator 680
Класс back_insert_iterator 681
Класс front_insert_iterator 682
Класс reverse_iterator 682
Класс istream_iterator 683
Класс istreambuf_iterator 684
Класс ostream_iterator 684
Класс ostreambuf_iterator 685
Две функции для работы с итераторами 686
Функторы 686
Функторы 686
Редакторы связей 687
Инверторы 688
Адаптеры 689
Адаптеры указателей на функции 689
Адаптеры указателей на функции — члены класса 690
Распределители памяти 691

Глава 36. Класс string 693

Класс basic_string 694
Класс char_traits 701

Глава 37. Числовые классы 703

Класс complex 704
Класс vallaray 706
Классы slice и gslice 716
Вспомогательные классы 717
Числовые алгоритмы 718
Класс accumulate 718
Алгоритм adjacent_difference 718
Алгоритм inner_product 719
Алгоритм partial_sum 720

Глава 38. Обработка исключительных ситуаций и прочие классы 723

Исключительные ситуации 724
Заголовок <exception> 724
Заголовок <stdexcept> 725
Класс auto_ptr 726
Класс pair 727
Локализация 728
Прочие классы 728

ЧАСТЬ V. ПРИЛОЖЕНИЯ НА ЯЗЫКЕ C++ 729

Глава 39. Интеграция новых классов: пользовательский класс для работы со строками 731

Класс StrType 732
Ввод и вывод строк 735
Функции присваивания 735
Конкатенация 737

Вычитание подстроки 738
Операторы сравнения 740
Прочие строковые функции 741
Полное определение класса StrType 741
Применение класса StrType 748
Принципы создания и интеграции новых типов 750
Проблема 750

Глава 40. Синтаксический анализ выражений 751

Выражения 752
Синтаксический анализ выражений: постановка задачи 753
Синтаксический анализ выражения 753
Класс parser 755
Разбор выражения на составные части 755
Простая программа синтаксического анализа выражений 758
Принципы работы синтаксического анализатора 762
Синтаксический анализатор, работающий с переменными 763
Проверка синтаксических ошибок при рекурсивном нисходящем анализе 770
Создание обобщенного синтаксического анализатора 771
Некоторые задачи 777

Приложение А. Расширение языка C++ для платформы .NET 779

Дополнительные ключевые слова 780
Ключевое слово __abstract 780
Ключевое слово __box 780
Ключевое слово __delegate 781
Ключевое слово __event 781
Ключевое слово __finally 781
Ключевое слово __gc 781
Ключевое слово __identifier 781
Ключевое слово __interface 781
Ключевое слово __nogc 781
Ключевое слово __pin 781
Ключевое слово __property 781
Ключевое слово __sealed 782
Ключевое слово __try_cast 782
Ключевое слово __typeof 782
Ключевое слово __value 782
Расширение директив препроцессора 782
Атрибут attribute 782
Компилирование управляемых программ на языке C++ 782

Приложение Б. Язык C++ и робототехника 783

Предметный указатель 787

Об авторе

Герберт Шилдт — автор наиболее популярных книг по программированию. Он является признанным специалистом по языкам C, C++, Java и C#, а также по программированию в среде Windows. Книги, написанные Шилдтом, переведены на все основные языки мира. Общий объем их продаж превышает 3 миллиона экземпляров. Он является автором многочисленных бестселлеров, в частности: *C++: The Complete Reference, C#: The Complete Reference, Java 2: The Complete Reference, C: The Complete Reference, C++ from the Ground Up, C++: A Beginner's Guide, C#: A Beginner's Guide* и *Java 2: A Beginner's Guide*. Шилдт получил звание магистра по компьютерным наукам в Университете штата Иллинойс (University of Illinois). Его консультацию можно получить по телефону офиса: (217) 586-4683.

Введение

Язык C++ оказывает огромное влияние на все области современного программирования. Его синтаксис, стиль и основные принципы стали стандартом при разработке новых языков. Благодаря своей универсальности он все чаще используется при описании алгоритмов и технологий программирования. Долговременный успех языка C++ определил основные тенденции развития компьютерных языков. Например, языки Java и C# являются его непосредственными наследниками. Скажем прямо, невозможно стать профессиональным программистом, не овладев языком C++. Это единственный язык, игнорировать который программист не имеет права.

Перед вами перевод четвертого издания книги *C++: The Complete Reference*. Здесь подробно описаны и проиллюстрированы все ключевые слова, функции, классы и свойства языка C++. Точнее говоря, в книге рассматривается вариант языка C++ под названием *Standard C++*. Именно эта версия соответствует международному стандарту ANSI/ISO и поддерживается всеми основными компиляторами, включая Visual C++ компании Microsoft и C++ Builder компании Borland. Таким образом, информацию, изложенную в книге, можно использовать во всех современных средах программирования.

За время, прошедшее после выхода в свет предыдущего издания книги, язык C++ не претерпел никаких изменений. В то же время радикально изменилась сама среда программирования. Например, появился новый стандарт языка C, получивший название *C99*, язык Java захватил господствующее положение в области Web-программирования, приобретает все большую популярность среда программирования .NET Framework. В потоке всех изменений, происшедших за последние годы, осталось незыблемым лишь одно — мощь языка C++. Его преимущество над другими языками в области разработки высокоэффективного программного обеспечения по-прежнему бесспорно.

Изменения, внесенные в четвертое издание

Общая структура справочника осталась прежней. Если вы читали третье издание, то сможете легко ориентироваться и в четвертом. Большинство изменений касается стиля изложения: в некоторых случаях добавлены новые подробности, в других местах иначе раскрыта тема. Кроме того, мы стремились учесть изменения, касающиеся современной среды программирования. В книгу также добавлено несколько новых разделов, например, в первой части иллюстрируется взаимосвязь между языком C++ и новым стандартом языка C, получившим название C99.

Книга содержит два новых приложения. В первом описаны расширенные ключевые слова, предложенные компанией Microsoft для создания управляемого кода по технологии .NET Framework. Второе приложение отражает личные вкусы автора и посвящено роботам, которые долгое время были его хобби. Надеемся, что у читателей вызовет интерес разработанный автором экспериментальный робот. Разумеется, большая часть программного обеспечения, управляющего этим роботом, написана на языке C++!

В заключение отметим, что все программы были заново перепроверены на современных компиляторах, включая компиляторы Visual Studio .NET компании Microsoft и C++ Builder компании Borland.

Содержание книги

В книге подробно освещены все аспекты языка C++, включая его основу — язык C. Справочник состоит из пяти частей.

- Подмножество C — основа языка C++.
- Язык C++.

- Библиотека стандартных функций.
- Библиотека стандартных классов.
- Приложения, написанные на языке C++.

Первая часть содержит подробное обсуждение языка C, который представляет собой подмножество языка C++. Как известно, язык C — это фундамент, на котором построен язык C++. Именно из языка C позаимствованы основные конструкции языка C++, например, циклы `for` и оператор `if`. Кроме того, язык C определил основные свойства блочных структур, указателей и функций в языке C++. Поскольку многие читатели уже хорошо знакомы с языком C, обсуждение его особенностей в самом начале книги позволит опытным программистам избежать повторения пройденного и перейти непосредственно к темам, посвященным собственно языку C++.

Во второй части справочника обсуждение выходит за рамки языка C и посвящается особенностям языка C++. Здесь рассматриваются его объектно-ориентированные свойства, например, классы, конструкторы, деструкторы, механизм динамической идентификации типа (Run-Time Type Identification — RTTI) и шаблоны. Таким образом, в части II описаны те конструкции, которые определяют сущность языка C++.

Третья часть посвящена стандартной библиотеке функций, а четвертая —классов, включая стандартную библиотеку шаблонов STL (Standard Template Library). В пятой части приведены два практических приложения, созданных с помощью языка C++ и объектно-ориентированного программирования.

Рекомендуемая литература

Полный справочник по C++ открывает перед читателями мир интереснейших книг по программированию, написанных Гербертом Шилдтом (Herbert Schildt).

Для дальнейшего изучения языка C++ обратите внимание на следующие издания.

C++: A Beginner's Guide
C++ from the Ground Up
Teach Yourself C++
STL Programming from the Ground Up
C++ Programmer's Reference

Для освоения языка Java мы рекомендуем такие книги.

Java 2: A Beginner's Guide
Java 2: The Complete Reference
Java 2: Programmer's Reference

О языке C# Герберт Шилдт написал следующие справочники.

C#: A Beginner's Guide
C#: The Complete Reference

При изучении программирования в среде Windows вам помогут такие книги.

Windows 98 Programming from the Ground Up
Windows 2000 Programming from the Ground Up
MFC Programming from the Ground Up
The Windows Programming Annotated Archives

Если вы захотите освоить язык C, лежащий в основе всего современного программирования, обязательно прочтите следующие издания.

C: The Complete Reference
Teach Yourself C

Если вам нужен быстрый и правильный ответ на ваш вопрос, обратитесь к Герберту Шилдту, признанному авторитету в области программирования.

Полный справочник по C++

Часть I

Основы языка C++: подмножество C

Описание языка C++ в книге разделено на две части. В части I обсуждаются свойства языка C, унаследованные языком C++. Обычно их называют подмножеством C языка C++. В части II рассматриваются специфические особенности языка C++. В совокупности эти две части содержат полное описание языка C++.

Вероятно, читателям уже известно, что язык C++ создан на основе языка C. Фактически язык C++ включает в себя весь язык C, и все программы (за некоторым исключением), написанные на языке C, можно считать программами на языке C++. В процессе разработки языка C++ в качестве отправной точки был выбран язык C. Затем к нему были добавлены новые свойства и возможности, разработанные для поддержки объектно-ориентированного программирования. Однако от языка C при этом не отказались, и стандарт 1989 года ANSI/ISO C Standard стал *базовым документом* при создании международного стандарта языка C++ (International Standard). Таким образом, осваивая язык C++, программисты одновременно овладевают языком C.

Разделение свойств языка C и специфических особенностей языка C++ позволяет достичь трех основных целей.

- Четко провести разделительную линию между языками C и C++.
- Предоставить читателям, владеющим языком C, возможность легко усвоить информацию об особенностях языка C++.
- Выделить и подробно описать свойства языка C++, унаследованные от языка C.

Очень важно точно провести линию, отделяющую язык C от языка C++, поскольку они очень широко распространены, и от программиста иногда требуется поддержка программ, написанных на обоих языках. Если вы пишете программу на языке C, нужно четко понимать, где заканчивается язык C и начинается язык C++. Многие программисты, работающие на языке C++, иногда пишут программы на языке C и не используют специфические свойства языка C++. В особенности это относится к программированию встроенных систем и поддержке существующих приложений. Таким образом, понимание различий между этими языками является необходимым условием профессионального программирования на C++.

Полное и свободное владение языком C абсолютно необходимо для перевода программ на язык C++. Чтобы сделать это на высоком уровне, необходимо хорошо знать язык C. Например, без ясного понимания механизмов ввода-вывода, предусмотренных в языке C, невозможно трансформировать программу, осуществляющую интенсивный обмен данными с внешними устройствами, в эффективную программу на языке C++.

Многие читатели уже владеют языком C. Вследствие этого выделение тем, связанных с языком C, позволяет опытным программистам избежать повторения пройденного и перейти непосредственно к изучению особенностей языка C++. Разумеется, в части I подчеркиваются малейшие отличия языка C++ от языка C. Кроме того, отделение подмножества C от остальных свойств языка C++ позволяет в дальнейшем сосредоточиться на его объектно-ориентированных особенностях.

Несмотря на то что язык C++ полностью содержит язык C, как правило, не все свойства языка C используются в программах, написанных "в стиле C++". Например, система ввода-вывода, предусмотренная в языке C, по-прежнему доступна в языке C++, хотя в C++ существуют свои объектно-ориентированные механизмы ввода-вывода данных. Еще одним примером такого анахронизма является препроцессор. Он играет чрезвычайно важную роль в языке C и очень скромную — в языке C++. Обсуждение свойств, присущих "стилю языка C", в первой части книги позволяет избежать путаницы в остальных главах.

Запомните: подмножество C, описанное в части I, является ядром языка C++ и фундаментом, на котором воздвигнуты его объектно-ориентированные конструкции. Все свойства, описанные здесь, являются неотъемлемой частью языка C++ и могут быть использованы в ваших программах.

На заметку

Часть I содержит материалы из моей книги C: The Complete Reference *(McGrawHill-Osborne) (Русский перевод: Г. Шилдт. Полный справочник по языку* C. *— М.: Изд. дом "Вильямс", 2001. — Прим. ред.). Если язык C интересует вас как самостоятельный язык программирования, эта книга окажется для вас ценным помощником.*

Полный справочник по

Глава 1

Обзор языка C

Чтобы понять язык C++, необходимо понять мотивы его создания, идеи, положенные в его основу, и свойства, которые он унаследовал от своих предшественников. Таким образом, история языка C++ начинается с языка C. В главе представлен обзор языка C, а также описана история его возникновения, способы применения и основные принципы. Поскольку язык C++ создан на основе языка C, эту главу можно считать описанием предыстории языка C++. Многое из того, что сделало язык C++ таким популярным, уходит корнями в язык C.

Происхождение и история языка C

Язык C был изобретен и впервые реализован Деннисом Ритчи (Dennis Ritchie) на компьютере DEC PDP–11 под управлением операционной системы UNIX. Язык C появился в результате развития языка под названием BCPL. В свою очередь, этот язык был разработан Мартином Ричардсом (Martin Richards) под влиянием другого языка, имевшего название B, автором которого был Кен Томпсон (Ken Tompson). Итак, в 1970-х годах развитие языка B привело к появлению языка C.

Многие годы фактическим стандартом языка C была версия для операционной системы UNIX. Впервые она была описана в книге Брайана Кернигана (Brian Kernigan) и Денниса Ритчи *The C Programming Language* в 1978 году. (Русский перевод: Керниган Б., Ритчи Д. Язык программирования C. — СПб: Невский диалект, 2001. — *Прим. ред.*). Летом 1983 года был создан комитет Американского института национальных стандартов (American National Standards Institute — ANSI), целью которого была разработка стандарта языка C. Работа комитета неожиданно растянулась на шесть лет.

В итоге стандарт ANSI C был одобрен в декабре 1989 года и стал распространяться в начале 1990-го. Этот стандарт был также одобрен Организацией международных стандартов (International Standards Organization — ISO), получив название *ANSI/ISO Standard C*. В 1995 году была одобрена Первая поправка, которая помимо всего прочего добавила несколько новых библиотечных функций. В 1989 году стандарт языка C вместе с Первой поправкой стали *базовым документом* для стандарта языка C++, в котором было выделено *подмножество C*. Версию языка C, определенную стандартом 1989 года, обычно называют *C89*.

После 1989 года в центре внимания программистов оказался язык C++. Развитие этого языка на протяжении 1990-х годов завершилось одобрением стандарта в конце 1998 года. Между тем работа над языком C продолжалась, не вызывая излишнего шума. В итоге в 1999 году появился новый стандарт языка C, который обычно называют *C99*. В целом стандарт C99 сохранил практически все свойства стандарта C89, не изменив основных аспектов языка. Таким образом, язык C, описанный стандартом C99, практически совпадает с языком, соответствующим стандарту C89. Комитет по разработке стандарта C99 сосредоточился на двух вопросах: включении в язык нескольких математических библиотек и развитии некоторых специфических и весьма сложных свойств, например, массивов переменной длины и квалификатора указателей `restrict`. В стандарт C99 вошли также некоторые свойства, позаимствованные из языка C++, например, однострочные комментарии. Поскольку разработка стандарта языка C++ завершилась до создания стандарта C99, ни одно из новшеств языка C не вошло в стандарт C++.

Сравнение стандартов C89 и C99

Несмотря на то что все новшества, внесенные в стандарт C99, весьма важны с теоретической точки зрения, они имели мало практических последствий, так как до сих пор нет ни одного широко распространенного компилятора, который

поддерживал бы стандарт C99. Большинство программистов до сих пор считают языком C его вариант, определенный стандартом C89. Именно его реализуют все основные компиляторы. Более того, подмножество C языка C++ описывается именно стандартом C89. Хотя некоторые новшества, включенные в стандарт C99, в конце концов обязательно будут учтены следующим стандартом языка C++, пока они несовместимы с языком C++.

Поскольку подмножество C языка C++ соответствует стандарту C89, и эту версию изучают большинство программистов, именно ее мы рассмотрим в части I. Итак, используя название C, мы будем иметь в виду версию языка, определенную стандартом C89. Однако мы будем отмечать важные различия между версиями C89 и C99, поскольку это улучшит совместимость языков C и C++.

C — язык среднего уровня

Язык C часто называют *языком среднего уровня*. Это не означает, что он менее эффективен, более неудобен в использовании или менее продуман, чем языки высокого уровня, такие как Basic или Pascal. Отсюда также не следует, что он запутан, как язык ассемблера (и порождает связанные с этим проблемы). Это выражение означает лишь, что язык C объединяет лучшие свойства языков высокого уровня, возможности управления и гибкость языка ассемблера. В табл. 1.1 показано место языка C среди других языков программирования.

Таблица 1.1. Место языка C среди остальных языков программирования

Высший уровень	Ada
	Modula-2
	Pascal
	COBOL
	FORTRAN
	BASIC
Средний уровень	Java
	C#
	C++
	C
	Forth
Низший уровень	Macro-assembler
	Assembler

Будучи языком среднего уровня, язык C позволяет осуществлять манипуляции с битами, байтами и адресами — основными элементами, с которыми работают функции операционной системы. Несмотря на это, программы, написанные на языке C, можно выполнять на разных компьютерах. Это свойство программ называется *машинонезависимостью* (portability). Например, если программу, написанную для операционной системы UNIX, можно легко преобразовать, чтобы она работала на платформе Windows, то говорят, что такая программа является *машинонезависимой* (portable).

Все языки высокого уровня используют концепцию типов данных. *Тип данных* (data type) определяет множество значений, которые может принимать переменная, а также множество операций, которые над ней можно выполнять. К основным типам данных относятся целое число, символ и действительное число. Несмотря на то что в языке C существует пять встроенных типов данных, он не является строго типизи-

рованным языком, как языки Pascal и Ada. В языке C разрешены практически все преобразования типов. Например, в одном и том же выражении можно свободно использовать переменные символьного и целочисленного типов.

В отличие от языков высокого уровня, язык C практически не проверяет ошибки, возникающие на этапе выполнения программ. Например, не осуществляется проверка возможного выхода индекса массива за пределы допустимого диапазона. Предотвращение ошибок такого рода возлагается на программиста.

Кроме того, язык C не требует строгой совместимости типов параметров и аргументов функций. Как известно, языки высокого уровня обычно требуют, чтобы тип аргумента точно совпадал с типом соответствующего параметра. Однако в языке C такого условия нет. Он позволяет использовать аргумент любого типа, если его можно разумным образом преобразовать в тип параметра. Кроме того, язык C предусматривает средства для автоматического преобразования типов.

Особенность языка C заключается в том, что он позволяет непосредственно манипулировать битами, байтами, машинными словами и указателями. Это делает его очень удобным для системного программирования, в котором эти операции широко распространены.

Другой важный аспект языка состоит в том, что в нем предусмотрено очень небольшое количество ключевых слов, которые можно использовать для конструирования выражений. Например, стандарт C89 содержит лишь 32 ключевых слова, а стандарт C99 добавил к ним всего 5 слов. Некоторые языки программирования содержат в несколько раз больше ключевых слов. Скажем, самые распространенные версии языка BASIC предусматривают более 100 таких слов!

C – структурированный язык

Возможно, вы уже слышали словосочетание *блочно-структурированный* (block-structured) по отношению к языку программирования. Хотя этот термин нельзя напрямую применять к языку C, его обычно тоже называют *структурированным*. Он имеет много общего с другими структурированными языками, такими как ALGOL, Pascal и Modula-2.

На заметку *Язык C (как и C++) не считается блочно-структурированным, поскольку не позволяет объявлять одну функции внутри других.*

Отличительной особенностью структурированных языков является *обособление* кода и данных (compartmentalization). Оно позволяет выделять и скрывать от остальной части программы данные и инструкции, необходимые для решения конкретной задачи. Этого можно достичь с помощью подпрограмм (subroutines), в которых используются локальные (временные) переменные. Используя локальные переменные, можно создавать подпрограммы, не порождающие побочных эффектов в других модулях. Это облегчает координацию модулей между собой. Если программа разделена на обособленные функции, нужно лишь знать, что делает та или иная функция, не интересуясь, как именно она выполняет свою задачу. Помните, что чрезмерное использование глобальных переменных (которые доступны в любом месте программы) повышает вероятность ошибок и нежелательных побочных эффектов. (Каждый программист, работавший на языке BASIC, хорошо знает эту проблему.)

На заметку *Концепция обособления широко применяется в языке C++. В частности, каждая часть программы, написанной на этом языке, может четко управлять доступом к ней из других модулей.*

Структурные языки предоставляют широкий спектр возможностей. Они допускают использование вложенных циклов, например **while**, **do-while** и **for**.

В структурированных языках использование оператора `goto` либо запрещено, либо нежелательно и не включается в набор основных средств управления потоком выполнения программы (как это принято в стандарте языка BASIC и в традиционном языке FORTRAN). Структурированные языки позволяют размещать несколько инструкций программы в одной строке и не ограничивают программиста жесткими полями для ввода команд (как это делалось в старых версиях языка FORTRAN).

Рассмотрим несколько примеров структурированных и неструктурированных языков (табл. 1.2).

Таблица 1.2. Структурированные и неструктурированные языки программирования

Неструктурированные	*Структурированные*
FORTRAN	Pascal
BASIC	Ada
COBOL	Java
	C#
	C++
	C
	Modula-2

Структурированные языки считаются более современными. В настоящее время неструктурированность является признаком устаревших языков программирования, и лишь немногие программисты выбирают их для создания серьезных приложений.

На заметку *Новые версии старых языков программирования (например Visual Basic) включают элементы структурированности. И все же врожденные недостатки этих языков вряд ли будут до конца исправлены, поскольку структурированность не закладывалась в их основу с самого начала.*

Основным структурным элементом языка C является *функция*. Именно функции служат строительными блоками, из которых создается программа. Они позволяют разбивать программу на модули, решающие отдельные задачи. Создав функцию, можно не беспокоиться о побочных эффектах, которые она вызовет в других частях программы. Способность создавать отдельные функции чрезвычайна важна при реализации больших проектов, в которых один фрагмент кода не должен взаимодействовать с другими частями программы непредсказуемым образом.

Другой способ структурирования и обособления программы, написанной на языке C, — блоки. *Блок* (code block) — это группа операторов, логически связанных между собой, и рассматриваемых как единое целое. В языке C блок можно создать с помощью фигурных скобок, ограничивающих последовательность операторов. Вот типичный пример блока.

```
if (x < 10) {
  printf("Слишком мало, попробуйте снова.\n";
  scanf("%d", &x);
}
```

Два оператора, расположенных внутри фигурных скобок, выполняются, если значение переменной `x` меньше 10. Эти два оператора вместе с фигурными скобками образуют блок. Блок — это логическая единица, поскольку оба оператора обязательно должны быть выполнены. Блоки позволяют ясно, элегантно и эффективно реализовывать различные алгоритмы. Более того, они помогают программисту лучше выразить природу алгоритма.

C — язык для программистов

Как ни странно, не все языки программирования предназначены для программистов. Рассмотрим классические примеры языков, ориентированных не на программистов, — COBOL и BASIC. Язык COBOL был разработан вовсе не для того, чтобы облегчить участь программистов, улучшить ясность кода и повысить его надежность, и даже не для того, чтобы ускорить выполнение программ. Он был создан, в частности, для того, чтобы люди, не являющиеся программистами, могли читать и (по возможности) понимать (хотя это вряд ли) написанные на нем программы. В свою очередь, язык BASIC был разработан для пользователей, которые решают на компьютере простые задачки.

В противоположность этому, язык C был создан для программистов, учитывал их интересы и многократно проверялся на практике, прежде чем был окончательно реализован. В итоге этот язык дает программистам именно то, чего они желали: сравнительно небольшое количество ограничений, минимум претензий, блочные структуры, изолированные функции и компактный набор ключевых слов. Язык C обладает эффективностью ассемблера и структурированностью языков ALGOL или Modula-2. Поэтому неудивительно, что именно языки C и C++ стали наиболее популярными среди профессиональных программистов высокого уровня.

Тот факт, что язык C часто используют вместо ассемблера, является одной из основных причин его популярности. Язык ассемблера использует символьное представление фактического двоичного кода, который непосредственно выполняется компьютером. Каждая операция, выраженная на языке ассемблера, представляет собой отдельную задачу, выполняемую компьютером. Хотя язык ассемблера предоставляет программисту наибольшую гибкость, разрабатывать и отлаживать программы на нем довольно сложно. Кроме того, поскольку язык ассемблера является неструктурированным, код напоминает спагетти — запутанную смесь переходов, вызовов и индексов. Вследствие этого программы, написанные на языке ассемблера, трудно читать, модифицировать и эксплуатировать. Вероятно, основным недостатком программ на языке ассемблера является их машинозависимость. Программа, предназначенная для конкретного центрального процессора, не может выполняться на компьютерах другого типа.

Изначально язык C предназначался для системного программирования. *Системная программа* (system program) представляет собой часть операционной системы или является одной из ее утилит. Рассмотрим некоторые из них.

- Операционные системы
- Интерпретаторы
- Редакторы
- Компиляторы
- Файловые утилиты
- Оптимизаторы
- Диспетчеры реального времени
- Драйверы

По мере роста популярности языка C многие программисты стали применять его для программирования всех задач, используя его машинонезависимость и эффективность, а кроме того, он им просто нравился! К моменту появления языка C языки

программирования прошли сложный и трудный путь совершенствования. Разумеется, вновь созданный язык вобрал в себя все лучшее.

С появлением языка C++ некоторые программисты посчитали, что язык C потеряет самостоятельность и сойдет со сцены. Однако этого не произошло. Во-первых, не все программы должны быть объектно-ориентированными. Например, программное обеспечение встроенных систем по-прежнему создается на языке C. Во-вторых, существует огромное множество программ на языке C, которые активно эксплуатируются и нуждаются в модификации. Поскольку язык C является основной языка C++, он продолжает широко использоваться, имея блестящие перспективы.

Структура программы на языке C

В табл. 1.3 перечислены 32 ключевых слова, которые используются при формировании синтаксиса языка C, стандарта C89 и подмножества C языка C++. Все они, конечно, являются и ключевыми словами языка C++.

Таблица 1.3. Ключевые слова подмножества C языка C++

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Кроме того, многие компиляторы для более эффективного использования среды программирования вносят в язык C дополнительные ключевые слова. Например, некоторые компиляторы предусматривают ключевые слова для управления памятью процессоров семейства 8086, поддержки многоязычного программирования и доступа к системным прерываниям. Перечислим некоторые из этих расширенных ключевых слов.

```
asm         _cs         _ds         _es
_ss         cdecl       far         huge
interrupt   near        pascal
```

Ваш компилятор может изменить этот список, стремясь наиболее эффективно использовать конкретную среду программирования.

Обратите внимание на то, что все ключевые слова набраны строчными буквами. Язык C/C++ *чувствителен к регистру* (case sensitive), т.е. прописные и строчные буквы в нем различаются. Это значит, что слово **`else`** является ключевым, а слово **`ELSE`** — нет. Ключевые слова нельзя использовать в программе для иных целей, например, в качестве имени переменной или функции.

Все программы на языке C состоят из одной или нескольких функций. В любом случае программа должна содержать функцию **`main()`**, которая при выполнении программы вызывается первой. В хорошо написанном коде функция **`main()`** должна содержать, по существу, схему работы всей программы. Несмотря на то что имя **`main()`** не включено в список ключевых слов, по своей природе оно является именно таковым. Скажем, назвать переменную именем **`main`** нельзя, так как компилятор сразу выдаст сообщение об ошибке.

Общий вид программы на языке C показан в листинге 1.1. Функции с именами `f1()`, ..., `fN()` определяются пользователем.

Листинг 1.1. Общий вид программы на языке C

```
объявления глобальных переменных
тип_возвращаемого_значения main (список параметров)
{
   последовательность операторов
}

тип_возвращаемого_значения f1 (список параметров)
{
   последовательность операторов
}

тип_возвращаемого_значения f2 (список параметров)
{
   последовательность операторов
}

.
.
.

тип_возвращаемого_значения fN (список параметров)
{
   последовательность операторов
}
```

Библиотека и связывание

С формальной точки зрения можно написать законченную и вполне осмысленную программу на языке C, не используя ни одной стандартной функции. Однако это довольно затруднительно, так как в языке C нет ключевых слов, имеющих отношение к вводу-выводу, математическим операциям высокого уровня или обработке символов. В результате большинство программ вынуждены вызывать различные функции, содержащиеся в *стандартной библиотеке* (standard library).

Все компиляторы языка C++ сопровождаются стандартной библиотекой функций, выполняющих наиболее распространенные задачи. В этой библиотеке определен минимальный набор функций, которые поддерживаются большинством компиляторов. Однако конкретный компилятор может содержать много других функций. Например, стандартная библиотека не содержит графических функций, но в каждом компиляторе всегда есть определенный набор таких функций.

Стандартная библиотека языка C++ разделена на две части: функции и классы. Стандартная библиотека функций представляет собой наследие языка C. Язык C++ поддерживает все функции, предусмотренные стандартом C89. Таким образом, все стандартные функции языка C можно свободно использовать в программах на языке C++.

Кроме стандартной библиотеки функций, язык C++ имеет собственную библиотеку классов. Эта библиотека состоит из объектно-ориентированных модулей, которые можно использовать в собственных программах. Кроме того, существует стандартная

библиотека шаблонов STL, содержащая широкий набор готовых решений многих задач. В части I используется только стандартная библиотека функций, поскольку именно она относится к языку C.

Стандартная библиотека состоит из многих универсальных функций. При вызове библиотечной функции компилятор "запоминает" ее имя. Позднее редактор связей объединит ваш код с объектным кодом этой библиотечной функции. Этот процесс называется *редактированием связей* (linking). Некоторые компиляторы имеют свои собственные редакторы связей, остальные используют редактор связей, предусмотренный операционной системой.

Функции в библиотеке имеют *машинонезависимый* формат (relocatable format). Это значит, что адреса памяти для разных мащинных инструкций не являются абсолютными — сохраняется лишь информация о смещении их адреса. Фактические адреса ячеек, используемых стандартными функциями, определяются во время редактирования связей. Подробное описание этих процессов можно найти в соответствующих технических руководствах. Более детальные объяснения были бы излишни.

В стандартной библиотеке содержится много функций, которые могли бы оказаться полезными. Они представляют собой крупные строительные блоки, из которых можно сконструировать свою программу. В частности, если какую-то функцию вы используете в своих программах очень часто, ее следует поместить в библиотеку.

Раздельная компиляция

Большинство коротких программ обычно можно уместить в одном файле. Однако по мере возрастания объема программы увеличивается время ее компилирования. Для решения этой проблемы в языке C/C++ предусмотрена возможность делить программу на файлы и компилировать каждый из них отдельно. Скомпилировав все файлы, отредактировав связи между ними и библиотечными функциями, вы получите завершенный объектный код. Преимущество раздельной компиляции заключается в том, что при изменении кода, записанного в одном из файлов, нет необходимости компилировать заново всю программу. Это существенно экономит время на этапе компиляции. Документация, сопровождающая компиляторы языка C/C++, содержит инструкции, которые позволят вам скомпилировать программу, записанную в нескольких файлах.

Расширения файлов .C и .CPP

Разумеется, программы, приведенные в части I, являются вполне корректными программами на языке C++. Их можно компилировать с помощью любого современного компилятора языка C++. Одновременно они являются корректными программами на языке C и могут компилироваться с помощью компиляторов этого языка. Итак, если вы собираетесь писать программы на языке C, можете рассматривать программы из первой части в качестве примера. Традиционно программы на языке C используют расширения файла **.C**, а программы на языке C++ — **.CPP**. Компилятор языка C++ использует расширение файла для определения типа программы. Это имеет большое значение, поскольку компилятор рассматривает любую программу, использующую расширение **.C**, как программу на языке C, а программу, записанную в файл с расширением **.CPP**, — как программу на языке C++. Если обратное не указано явно, вы можете выбрать любое расширение для файла с программой из первой

части книги. Однако программы, приведенные в остальной части книги, должны быть записаны в файлы с расширением **.CPP**.

И последнее: хотя язык C является подмножеством языка C++, между ними существует несколько отличий. В некоторых случаях программу на языке C нужно компилировать *именно как программу на языке C* (используя расширение **.C**). Все такие случаи мы оговариваем отдельно.

Полный справочник по C++

Глава 2

Выражения

В этой главе рассматриваются самые важные элементы языка C (и, соответственно, языка C++) — *выражения* (expressions). В языке C/C++ выражения носят намного более абстрактный характер, чем в большинстве других языков программирования, и обладают большей экспрессивностью. Выражения состоят из атомарных элементов: данных и операторов. Данные представляют собой либо переменные, либо константы. Как и во многих языках программирования, в языке C/C++ предусмотрено большое количество разнообразных типов данных и операторов.

Пять основных типов данных

В подмножестве C существуют пять элементарных типов данных: символ, целое число, число с плавающей точкой, число с плавающей точкой удвоенной точности и переменная, не имеющая значений. Им соответствуют следующие ключевые слова: **char**, **int**, **float**, **double** и **void**. Все другие типы данных в языке C создаются на основе элементарных типов, указанных выше. Размер переменных и диапазон их значений зависит от типа процессора и компилятора. Однако во всех случаях размер символа равен 1 байт. Размер целочисленной переменной обычно равен длине машинного слова, принятой в конкретной операционной системе. В большинстве 16-битовых операционных систем, например, DOS и Windows 3.1, размер целочисленной переменной равен 16 бит. В большинстве 32-битовых операционных систем, например Windows 2000, этот размер равен 32 бит. Однако, стремясь к машинонезависимости программ, следует избегать конкретных предположений о размере целочисленных переменных. Важно четко понимать, что и в языке C, и в языке C++ оговаривается лишь *минимальный диапазон*, в котором изменяются значения переменных каждого типа, а не их размер в байтах.

На заметку *К пяти основным типам данных, определенных в языке C, язык C++ добавляет еще два: **bool** и **wchar_t**. Эти типы будут рассмотрены во второй части книги.*

Точное представление чисел с плавающей точкой зависит от их конкретной реализации. Размер целого числа обычно равен длине машинного слова, принятой в операционной системе. Значения переменных типа **char**, как правило, используются для представления символов, предусмотренных в системе кодирования ASCII. Значения, выходящие за пределы допустимого диапазона, на разных компьютерах обрабатываются по-разному.

Диапазон изменения переменных типа **float** и **double** зависит от способа представления чисел с плавающей точкой. В любом случае, этот диапазон достаточно широк. Стандарт языка C определяет минимальный диапазон изменения чисел с плавающей точкой: от 1E–37 до 1E+37. Минимальное количество цифр, определяющих точность чисел с плавающей точкой, указано в табл. 2.1.

Таблица 2.1. Все типы данных, определенные в стандарте ANSI/ISO C Standard

Тип	*Обычный размер, бит*	*Минимальный диапазон*
char	8	От –128 до 127
unsigned char	8	От 0 до 255
signed char	8	От –128 до 127
int	16 или 32	От –32768 до 32767
unsigned int	16 или 32	От 0 до 65535
signed int	16 или 32	Такой же, как у **int**
short int	16	От –32768 до 32767
unsigned short int	16	От 0 до 65535

Окончание табл. 2.1

Тип	*Обычный размер, бит*	*Минимальный диапазон*
`signed short int`	16	Такой же, как и у `short int`
`long int`	32	От –2147483648 до 2147483647
`signed long int`	32	Такой же, как и у `long int`
`unsigned long int`	32	От 0 до 4294967295
`float`	32	Шесть значащих цифр
`double`	64	Десять значащих цифр
`long double`	80	Десять значащих цифр

На заметку

В стандарте языка C++ минимальный размер и диапазон изменения переменных, имеющих элементарный тип, не определен. Вместо этого в нем просто указано, что они должны соответствовать определенным условиям. Например, стандарт требует, чтобы переменная типа `int` *"имела естественный размер, соответствующий архитектуре операционной системы". В любом случае, ее диапазон должен совпадать или превосходить диапазон изменения переменной данного типа, предусмотренный стандартом языка C. Каждый компилятор языка C++ задает размер и диапазон изменения переменных всех элементарных типов в заголовке* `<climits>`.

Тип `void` используется для определения функции, не возвращающей никаких значений, либо для создания обобщенного указателя (generic pointer). Оба эти случая мы рассмотрим в следующих главах.

Модификация основных типов

За исключением типа `void`, основные типы данных могут иметь различные *модификаторы* (modifiers), которые используются для более точной настройки. Вот их список.

```
signed
unsigned
long
short
```

Целочисленные типы можно модифицировать с помощью ключевых слов `signed`, `short`, `long` и `unsigned`. Символьные типы можно уточнять с помощью модификаторов `unsigned` и `signed`. Кроме того, тип `double` можно модифицировать ключевым словом `long`. В табл. 2.1 указаны все возможные комбинации типов данных, а также их минимальные диапазоны и приблизительный размер в битах. (Эти значения относятся также и к языку C++.) Учтите, что в этой таблице приведены *минимальные диапазоны* переменных, а не типичные. Например, в компьютерах, использующих арифметику дополнительных кодов (two's complement arithmetic), минимальный диапазон целого числа простирается от −32768 до 32767.

Применение модификатора `signed` допускается, но является излишним, поскольку по умолчанию все целые числа имеют знак. Наиболее важен этот модификатор при уточнении типа `char` в тех реализациях языка C, где тип `char` по умолчанию знака не имеет.

Разница между целыми числами, имеющими и не имеющими знак, заключается в интерпретации бита в старшем разряде. Если в программе определено целое число со знаком, то компилятор генерирует код, в котором старший бит интерпретируется как *признак знака* (sign flag). Если признак знака равен 0, число считается положительным, если он равен 1 — отрицательным.

Иначе говоря, отрицательные числа представляются с помощью *дополнительного кода* (two's complement), в котором все биты числа (за исключением старшего разряда) инвертированы, затем к полученному числу добавляется единица, а признак знака устанавливается равным 1.

Целые числа со знаком играют весьма важную роль в очень многих алгоритмах, но диапазон их изменения вдвое короче, чем у целых чисел, не имеющих знака. Рассмотрим двоичное представление числа 32767.

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Допустим, старший бит равен 1, значит, это число станет равным −1. Однако, если бы эта переменная была объявлена как **unsigned int**, ее значение стало бы равным 65535.

Если модификатор типа используется отдельно (т.е. без указания элементарного типа), по умолчанию предполагается, что объявляемая переменная имеет тип **int**. Итак, указанные ниже спецификаторы типов эквивалентны.

Спецификатор	*Аналог*
`signed`	`signed int`
`unsigned`	`unsigned int`
`long`	`long int`
`short`	`short int`

Хотя одного спецификатора **int** вполне достаточно, многие программисты предпочитают явно указывать его модификации.

Идентификаторы

В языке C/C++ имена переменных, функций, меток и других объектов, определенных пользователем, называются *идентификаторами* (indentifiers). Идентификаторы могут состоять из одного или нескольких символов. Первый символ идентификатора должен быть буквой или символом подчеркивания, а следующие символы должны быть буквами, цифрами или символами подчеркивания. Ниже приведены примеры правильных и неправильных идентификаторов.

Правильно	*Неправильно*
`Count`	`1count`
`test23`	`hi!there`
`high_balance`	`high...balance`

В языке C длина идентификаторов может быть произвольной. Однако не все символы, образующие идентификатор, считаются значащими. Если идентификатор используется в процессе редактирования внешних связей, то значащими считаются, по крайней мере, шесть его первых символов. Эти идентификаторы называются *внешними именами* (external names). К ним относятся имена функций и глобальных переменных, используемых несколькими файлами. Если идентификатор не используется в процессе редактирования внешних связей, то значащими считаются, по крайней мере, 31 первый символ. Такие идентификаторы называются *внутренними именами* (internal names). К ним относятся, например, имена локальных переменных. В языке C++ нет ограничений на длину идентификаторов и значащими считаются, по крайней мере, 1024 первых символа. При переходе от языка с к языку C++ этот нюанс следует учитывать.

Символы, набранные в верхнем и нижнем регистре, различаются. Следовательно, **count**, **Count** и **COUNT** — это разные идентификаторы.

Ключевые слова нельзя использовать в качестве идентификаторов ни в языке C, ни в языке C++. Кроме того, идентификаторы не должны совпадать с именами функций из стандартных библиотек.

Переменные

Переменная (variable) представляет собой имя ячейки памяти, которую можно использовать для хранения модифицируемого значения. Все переменные должны быть объявлены до своего использования. Ниже приведен общий вид объявления переменной.

тип список_переменных;

Здесь слово *тип* означает один из допустимых типов данных, включая модификаторы, а *список_переменных* может состоять из одного или нескольких идентификаторов, разделенных запятыми. Рассмотрим несколько примеров объявлений.

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Учтите, в языке C/C++ имя переменной никак не связано с ее типом.

Где объявляются переменные

Как правило, переменные объявляют в трех местах: внутри функций, в определении параметров функции и за пределами всех функций. Соответственно такие переменные называются локальными, формальными параметрами и глобальными.

Локальные переменные

Переменные, объявленные внутри функции, называются *локальными* (local variables). В некоторых книгах, посвященных языку C/C++, эти переменные называются *автоматическими* (automatic variables). В нашей книге мы будем придерживаться более общей терминологии и называть их локальными. Локальные переменные можно использовать только в операторах, расположенных внутри блока, где они объявлены. Иначе говоря, локальные переменные невидимы снаружи их блока. Напомним, что блок ограничен открывающей и закрывающей фигурными скобками.

Чаще всего локальные переменные объявляются внутри функций. Рассмотрим два примера.

```
void func1(void)
{
 int x;

 x = 10;
}

void func2(void)
{
 int x;

 x = -199;
}
```

Переменная **x** объявлена дважды: сначала — в функции **finc1()**, а затем — в функции **func2()**. Переменная **x** из функции **func1()** не имеет никакого отношения

к переменной **x**, объявленной внутри функции **func2()**. Каждая из этих переменных существует только внутри блока, где она была объявлена.

Язык С содержит ключевое слово **auto**, которое можно использовать для объявления локальных переменных. Однако, поскольку все локальные переменные по умолчанию считаются автоматическими, это ключевое слово на практике никогда не используется. По этой причине в нашем справочнике мы также не будем его применять. (Говорят, что ключевое слово **auto** было включено в язык С для обеспечения совместимости с его предшественником — языком В. Затем это слово перекочевало в язык С++ для обеспечения совместимости с языком С.)

Из соображений удобства и по традиции большинство программистов объявляют все переменные, используемые в функции, сразу после открывающей фигурной скобки и перед всеми остальными операторами. Однако следует иметь в виду, что локальные переменные можно объявлять в любом месте блока. Рассмотрим следующий пример.

```
void f(void)
{
 int t;

 scanf("%d%*c", &t);

 if(t==1) {
 char s[80]; /* Эта переменная создается только
                при входе в данный блок */
 printf("Введите имя:");
 gets(s);
 /* Некие операции ... */
 }
}
```

В этом фрагменте локальная переменная **s** создается при входе в блок **if** и разрушается сразу после выхода из него. Кроме того, переменная **s** является видимой только внутри блока **if**, и к ней невозможно обратиться извне, даже из других частей функции, содержащей данный блок.

Объявление переменных внутри блоков позволяет избежать побочных эффектов. Поскольку локальная переменная вне блока не существует, ее значение невозможно изменить непреднамеренно.

Способы объявления локальных переменных в языке С (в соответствии со стандартом С89) и в языке С++ существенно отличаются друг от друга. В языке С все локальные переменные должны быть объявлены в начале блока до выполнения каких-либо "активных" операторов. Например, попытка объявить локальные переменные так, как показано в приведенном ниже фрагменте, вызовет ошибку.

```
/* Если эта функция является частью программы на языке С,
   при компиляции возникнет ошибка. Если эта функция
   представляет собой часть программы на языке С++,
   ошибок не будет.
*/

void f(void)
{
 int i;

 i = 10;

 int j; /* Этот оператор порождает ошибку */

 j = 20;
}
```

Между тем, в языке C++ эта функция считается абсолютно правильной, поскольку в этом языке локальные переменные можно объявлять в любом месте блока, но до их первого использования. (Более подробно эта тема будет рассматриваться во второй части книги.) Интересно, что стандарт C99 также позволяет объявлять локальные переменные где угодно.

Поскольку локальная переменная создается при входе в блок, где она объявлена, и разрушается при выходе из него, ее значение теряется. Это особенно важно учитывать при вызове функции. Локальные переменные функции создаются при ее вызове и уничтожаются при возврате управления в вызывающий модуль. Это значит, что между двумя вызовами функции значения ее локальных переменных не сохраняются. (Заставить их сохранять свои значения можно с помощью модификатора **static**.)

По умолчанию локальные переменные хранятся в стеке. Поскольку стек является динамической структурой, и объем занимаемой им памяти постоянно изменяется, становится понятным, почему локальные переменные в принципе не могут сохранять свои значения между двумя вызовами функции, внутри которой они объявлены.

Локальную переменную можно проинициализировать неким значением. Оно будет присваиваться переменной каждый раз при входе в блок. Рассмотрим в качестве примера программу, которая десять раз выводит на печать число 10.

```
#include <stdio.h>

void f(void);

int main(void)
{
 int i;

 for(i=0; i<10; i++) f();

 return 0;
}

void f(void)
{
 int j = 10;

 printf("%d ", j);

 j++; /* Этот оператор не имеет долговременного эффекта. */
}
```

Формальные параметры

Если функция имеет аргументы, следует объявить переменные, которые будут принимать их значения. Эти переменные называются *формальными параметрами* (formal parameters). Внутри функции они ничем не отличаются от других локальных переменных. Как показано в приведенном ниже фрагменте программы, объявление таких переменных должно размещаться сразу после имени функции и заключаться в скобки.

```
/* Функция возвращает 1, если символ c является частью
   строки s; в противном случае она возвращает 0 */
int is_in(char *s, char c)
{
 while(*s)
 if(*s==c) return 1;
 else s++;
```

```
 return 0;
}
```

Функция **is_in()** имеет два параметра: **s** и **c**. Она возвращает 1, если символ **c** является частью строки **s**, в противном случае функция возвращает 0.

Тип формальных параметров задается при их объявлении. После этого их можно использовать как обычные локальные переменные. Учтите, что, как и локальные переменные, формальные параметры являются динамическими и разрушаются при выходе из функции.

Формальные параметры можно использовать в любых конструкциях и выражениях. Несмотря на то что свои значения они получают от аргументов, передаваемых извне функции, во всем остальном они ничем не отличаются от локальных переменных.

Глобальные переменные

В отличие от локальных, *глобальные переменные* (global variables) доступны из любой части программы и могут быть использованы где угодно. Кроме того, они сохраняют свои значения на всем протяжении выполнения программы. Объявления глобальных переменных должны размещаться вне всех функций. Эти переменные можно использовать в любом выражении, в каком бы блоке оно не находилось.

В приведенном ниже фрагменте программы переменная **count** объявлена вне всех функций. Несмотря на то что ее объявление расположено до функции **main()**, эту переменную можно было бы с таким же успехом объявить в другом месте, но вне функций и до ее первого использования. И все же лучше всего размещать объявления глобальных переменных в самом начале программы.

```
#include <stdio.h>
int count; /* Переменная count является глобальной */

void func1(void);
void func2(void);

int main(void)
{
 count = 100;
 func1();

 return 0;
}

void func1(void)
{
 int temp;

 temp = count;
 func2();
 printf("count = %d", count); /* Выведет число 100. */
}

void func2(void)
{
 int count;

 for(count=1; count<10; count++)
 putchar('.');
}
```

Присмотритесь к этой программе повнимательнее. Заметьте, что, хотя ни функция **main()**, ни функция **func1()** не содержат объявления переменной **count**, обе эти функции успешно ее используют. Однако внутри функции **func2()** объявлена локальная переменная **count**. Когда функция **func2()** ссылается на переменную **count**, она использует лишь локальную переменную, а не глобальную. Если глобальная и локальная переменные имеют одинаковые имена, то все ссылки на имя переменной внутри блока будут относиться к локальной переменной и не влиять на ее глобальную тезку. Возможно, это удобно, однако об этой особенности легко забыть, и тогда действия программы могут стать непредсказуемыми.

Глобальные переменные хранятся в специально отведенном для них месте. Они оказываются весьма полезными, если разные функции в программе используют одни и те же данные. Однако, если в них нет особой необходимости, глобальных переменных следует избегать. Дело в том, что они занимают память во время всего выполнения программы, даже когда уже не нужны. Кроме того, использование глобальных переменных там, где можно было бы обойтись локальными, ослабляет независимость функций, поскольку они вынуждены зависеть от сущности, определенной в другом месте. Итак, применение большого количества глобальных переменных повышает вероятность ошибок вследствие непредсказуемых побочных эффектов. Большинство проблем, возникающих при разработке больших программ, является следствием непредвиденного изменения значений переменных, которые используются в любом месте программы. Это может случиться, если в программе объявлено слишком много глобальных переменных.

Квалификаторы const и volatile

В языке C существуют два квалификатора, управляющих доступом и модификацией: **const** и **volatile**. Их следует указывать перед модификаторами типов и именами типов, которые они уточняют. Формально эти квалификаторы называют *cv-квалификаторами* (cv-qualifiers).

Квалификатор const

Переменные типа **const** не могут изменяться, однако их можно инициализировать. Компилятор может поместить эти переменные в постоянное запоминающее устройство (Random Access Memory — ROM). Рассмотрим пример.

```
const int a=10;
```

Это объявление создает целочисленную переменную с именем **a** и начальным значением, равным числу 10, которое в остальной части программы изменить невозможно. Однако переменную **a** можно использовать в других выражениях. Переменная типа **const** может получить свое значение либо во время инициализации, либо с помощью других машинозависимых средств.

Квалификатор **const** можно использовать для защиты объектов, передаваемых в функцию в качестве аргументов. Иными словами, если в функцию передается указатель, она может модифицировать фактическое значение, на которое он ссылается. Но если указатель уточнить квалификатором **const**, функция не сможет изменить значение в соответствующей ячейке. Например, функция **sp_to_dash()**, рассмотренная ниже, выводит на печать тире, заменяя им пробелы в строке, которая является ее аргументом. Иначе говоря, "строка для проверки" будет напечатана как "строка–для–проверки". Использование квалификатора **const** в объявлении параметра гарантирует, что код, помещенный внутри функции, не сможет модифицировать объекты, на которые ссылается указатель.

```
#include <stdio.h>

void sp_to_dash(const char *str);

int main(void)
{
 sp_to_dash("строка для проверки");

 return 0;
}

void sp_to_dash(const char *str)
{
 while(*str) {
 if(*str== ' ') printf("%c", '-');
 else printf("%c", *str);
 str++;
 }
}
```

Если бы мы попытались модифицировать строку внутри функции **sp_to_dash()**, компилятор выдал бы сообщение об ошибке. Например, если написать функцию **sp_to_dash()** так, как показано ниже, то возникла бы ошибка компиляции.

```
/* Ошибка */
void sp_to_dash(const char *str)
{
 while(*str) {
 if(*str==' ' ) *str = '-'; /* Нельзя! Аргумент str
                               является константным */
 printf("%c", *str);
 str++;
 }
}
```

Многие функции из стандартной библиотеки используют квалификатор **const** в объявлениях своих параметров. Например, прототип функции **strlen()** выглядит следующим образом.

```
size_t strlen(const char *str);
```

Объявив указатель **str** константной, авторы защитили от изменений строку, на которую он ссылается. В принципе, если в стандартной функции нет необходимости модифицировать объект, на который ссылается ее аргумент, то этот аргумент объявляется константным.

Квалификатор **const** можно также использовать для предотвращения модификации переменных. Помните, что переменную типа **const** можно модифицировать лишь за пределами программы. Например, ее значение можно изменить с помощью аппаратных устройств. Однако, если переменная объявлена константной, можно гарантировать, что ее изменение может произойти лишь вследствие внешнего вмешательства.

Квалификатор volatile

Квалификатор **volatile** сообщает компилятору, что значение переменной может изменяться неявно. Например, адрес глобальной переменной можно передать таймеру операционной системы и использовать его для отсчета реального времени. В этом

случае содержимое переменной изменяется без явного выполнения какого-либо оператора присваивания. Это очень важно, поскольку большинство компиляторов языка C/C++ автоматически оптимизируют некоторые выражения, считая, что значения переменных не изменяются, если они не указаны в левой части оператора присваивания. Таким образом, их значения нет смысла перепроверять при каждом обращении. Кроме того, некоторые компиляторы изменяют порядок вычисления выражений в ходе компиляции. Квалификатор **volatile** предотвращает такие изменения.

Квалификаторы **const** и **volatile** можно использовать одновременно. Например, если 0x30 — значение, хранящееся в порте, которое изменяется только под влиянием внешних обстоятельств, то следующее объявление предотвратит непредвиденные побочные эффекты.

```
const volatile char *port = (const volatile char *) 0x30;
```

Спецификаторы хранения

В языке C существуют четыре спецификатора хранения.

extern
static
register
auto

Эти спецификаторы сообщают компилятору, как хранить объявленную переменную. Общий вид объявления, использующего эти спецификаторы, приведен ниже.

спецификатор_хранения тип имя_переменной

Обратите внимание на то, что спецификаторы хранения предшествуют всем остальным элементам объявления.

На заметку *В языке C++ предусмотрен еще один спецификатор хранения* **mutable**. *Он рассматривается в части II.*

Спецификатор extern

Прежде чем рассмотреть спецификатор **extern**, опишем систему связей в языке C/C++. В языке C/C++ предусмотрено три категории: внешние, внутренние связи и их отсутствие. Как правило, функции и глобальные переменные имеют внешние связи. Это означает, что они доступны из любой части программы. Глобальные объекты, объявленные с помощью спецификатора **static** (описанного в следующем разделе), имеют внутренние связи. Они доступны лишь внутри файла, в котором описаны. Локальные переменные не имеют связей и, следовательно, видимы лишь внутри своего блока.

Главное предназначение спецификатора **extern** — указать, что объявляемый объект обладает внешними связями в рамках всей программы. Чтобы понять, почему это так важно, необходимо различать *объявление* (declaration) и *определение* (definition). В объявлении указываются имя и тип объекта. Определение выделяет память для объекта. В программе может быть несколько объявлений одного и того же объекта, но только одно определение.

В большинстве случаев объявление переменной одновременно является ее определением. Однако, указав перед именем переменной спецификатор **extern**, можно объявить ее, не определяя. Таким образом, чтобы обратиться к переменной, определенной в другой части программы, ее следует объявить, используя спецификатор **extern**.

Рассмотрим пример использования спецификатора **extern**. Обратите внимание на то, что глобальные переменные **first** и **last** объявлены после функции **main()**.

```
#include <stdio.h>

int main(void)
{
 extern int first; /* Глобальные переменные */

 printf("%d %d", first, last);

 return 0;
 }

/* Глобальное определение переменных first и last */
int first = 10, last = 20;
```

Эта программа выведет на печать числа **10 20**, поскольку глобальные переменные **first** и **last**, которые используются при вызове функции **printf()**, инициализированы именно этими значениями. Поскольку спецификатор **extern** в функции **main()** сообщает компилятору, что переменные **first** и **last** объявлены в другом месте (в данном случае в конце текущего файла), программа будет скомпилирована без ошибок, даже если переменные **first** и **last** будут использованы раньше своих определений.

Следует иметь в виду, что объявление переменных в качестве внешних, как показано в предыдущей программе, необходимо лишь потому, что переменные **first** и **last** не были объявлены до своего использования в функции **main()**. В противном случае необходимости в спецификаторе **extern** не было бы. Обнаружив переменную, ранее не объявленную в текущем блоке, компилятор проверяет, не объявлена ли она в охватывающих блоках. Если нет, компилятор проверяет глобальные переменные. Обнаружив совпадение, компилятор считает, что данная ссылка относится к ранее объявленной переменной. Если необходимо использовать переменную, объявленную позднее в том же файле, следует применять спецификатор **extern**.

Как указывалось выше, спецификатор **extern** позволяет объявить переменную, не определяя ее. Однако, если при этом выполняется инициализация переменной, объявление **extern** становится определением. Это важно, поскольку объект может иметь несколько объявлений, но только одно определение.

Спецификатор **extern** играет важную роль в программах, состоящих из нескольких файлов. Программу, написанную на языке C/C++, можно разделить на несколько файлов, скомпилировать их отдельно, а затем отредактировать связи между ними. Для этого необходимо иметь возможность сообщить всем файлам о глобальных переменных, существуюших в программе. Лучше всего объявить все глобальные переменные в одном файле, а в других объявить их с помощью спецификатора **extern**, как показано ниже. Это усилит машинонезависимость программы.

Первый файл	*Второй файл*

`int x, y;`
`char ch;`
`int main(void)`
`{`
`/* ...*/`
`}`
 	
`void func1(void)`
`{`
`x = 123;`
`}`
	`extern int x, y;`
	`extern char ch;`
	`void func22(void)`
	`{`
	`x = y / 10;`
	`}`
	
	`void func23(void)`
	`{`
	`y = 10;`
	`}`

Список глобальных переменных копируется из первого файла во второй, а затем добавляется спецификатор **extern**. Он сообщает компилятору, что имена и типы переменных, следующих далее, объявлены в другом месте. Иными словами, спецификатор **extern** сообщает компилятору типы и имена глобальных переменных, не выделяя для них память еще раз. Все ссылки на внешние переменные распознаются в процессе редактирования связей.

На практике объявления внешних переменных обычно хранятся в заголовочных файлах, которые просто включаются в исходный код. Это и легче, и надежнее, чем дублировать все объявления внешних переменных в каждом файле вручную.

В языке C++ спецификатор **extern** имеет еще одно применение, которое описано в части II.

На заметку *Спецификатор* ***extern*** *можно также применять к объявлениям функций, но это излишне.*

Статические переменные

Статические переменные, в отличие от глобальных, неизвестны вне своей функции или файла, и сохраняют свои значения между вызовами. Это очень полезно при создании обобщенных функций и библиотек функций, которые могут использоваться другими программистами. Статические переменные отличаются как от локальных, так и от глобальных переменных.

Локальные статические переменные

Если локальная переменная объявлена с помощью спецификатора **static**, компилятор выделит для нее постоянное место хранения, как и для глобальной переменной. Принципиальное отличие локальной статической переменной от глобальной состоит в том, что первая остается доступной лишь внутри своего блока. Проще говоря, локальная статическая переменная — это локальная переменная, сохраняющая свои значения между вызовами функции.

Такие переменные очень полезны при создании изолированных функций, поскольку между вызовами их можно использовать в других частях программы. Если бы статических переменных не было, пришлось бы применять глобальные переменные, порождая неизбежные побочные эффекты. Примером удачного применения статических переменных является генератор чисел, который порождает новое число, используя предыдущее значение. Для хранения этого числа можно было бы использовать глобальную переменную, однако при каждом новом вызове ее пришлось бы объявлять заново, постоянно проверяя, не конфликтует ли она с другими глобальными переменными. Применение статической переменной легко решает эту проблему.

```
int series(void)
{
 static int series_num;

 series_num = series_num+23;
 return series_num;
}
```

В этом примере переменная **series_num** продолжает существовать между вызовами функции, а локальная переменная каждый раз создавалась бы при входе и уничтожалась при выходе из функции. Таким образом, каждый вызов функции **series()** порождает новый элемент ряда, используя предыдущее значение и не прибегая к глобальной переменной.

Локальную статическую переменную можно инициализировать. Начальное значение присваивается лишь один раз, а не при каждом входе в блок, как это происходит с локальными переменными. Например, в приведенной ниже версии функции **series()** переменная **series_num** инициализируется числом 100.

```
int series(void)
{
 static int series_num = 100;

 series_num = series_num+23;
 return series_num;
}
```

Теперь ряд будет всегда начинаться с числа 123. Хотя во многих приложениях это вполне приемлемо, обычно генераторы чисел предоставляют пользователю право выбора начального значения. Для этого можно сделать переменную **series_num** глобальной. Однако именно для того, чтобы избежать этого, и были созданы статические переменные. Это приводит ко второму способу использования статических переменных.

Глобальные статические переменные

Применение спецификатора **static** к глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в пределах текущего файла. Несмотря на то что эта переменная остается глобальной, в других файлах она не существует. Следовательно, изменить ее значение путем вмешательства извне невозможно. Это предотвращает побочные эффекты. В некоторых ситуациях, в которых локальные статические переменные оказываются недостаточными, можно создать небольшой файл, содержащий лишь функции, которые используют конкретную глобальную статическую переменную, отдельно скомпилировать его и применять без риска возникновения побочных эффектов.

Чтобы проиллюстрировать применение глобальной статической переменной, перепишем генератор чисел из предыдущего раздела таким образом, чтобы начальное значение задавалось при вызове функции **series_start()**. Весь файл, содержащий функции **series()**, **series_start()** и **series_num()**, показан ниже.

```
/* Все функции должны находиться в одном и том же файле. */

static int series_num;
void series_start(int seed);
int series(void);

int series(void)
{
 series_num = series_num+23;
 return series_num;
}

/* Инициализация переменной series_num */
void series_start(int seed)
{
 series_num = seed;
}
```

Вызов функции **series_start()** инициализирует генератор чисел. После этого следующий элемент ряда порождается новым вызовом функции **series()**.

К сведению: локальные статические переменные видимы лишь в пределах блока, где они объявлены, а глобальные статические переменные — в пределах файла. Если поместить функции **series()** и **series_start()** в библиотеку, то переменная

series_num станет невидимой. Более того, в программе можно объявить новую переменную **series_num** (разумеется, в другом файле). По существу, модификатор **static** позволяет создавать переменные, видимые лишь в пределах функций, не порождая побочных эффектов.

Модификатор **static** позволяет скрывать часть программы от других модулей. Это чрезвычайно важно при разработке больших и сложных программ.

На заметку

*В языке C++ эта особенность модификатора **static** сохраняется, но объявлена нежелательной. Это означает, что при разработке новых программ использовать это свойство не рекомендуется. Вместо него следует применять пространства имен, описанные в части II.*

Спецификатор register

Спецификатор **register** изначально применялся лишь к переменным типа **int**, **char** или указателям. Однако впоследствии его толкование было расширено, и теперь его можно применять к переменным любого типа.

Спецификатор **register** был разработан для того, чтобы компилятор сохранял значение переменной в регистре центрального процессора, а не памяти, как обычно. Это означает, что операции над регистровыми переменными выполняются намного быстрее.

В настоящее время смысл спецификатора **register** трактуется очень широко. В стандарте языка C утверждается, что "доступ к объекту следует максимально ускорить". (В стандарте языка C++ говорится, что спецификатор **register** — это "подсказка компилятору, что данный объект используется очень интенсивно".) На практике в регистрах центрального процессора сохраняются символы и целые числа. Более крупные объекты наподобие массивов, очевидно, там не поместятся, но они сохраняют возможность приоритетной обработки компилятором. В зависимости от реализации компилятора и операционной системы обработка регистровой переменной осуществляется по-разному. С технической точки зрения компилятор может просто игнорировать спецификатор **register** и обрабатывать переменную как обычно, но на практике это случается редко.

Спецификатор **register** можно применять только к локальным переменным и формальным параметрам. К глобальным переменным он не применяется. В рассмотренном ниже примере функция вычисляет значение M^e для целых чисел.

```
int int_pwr(register int m, register int e)
{
 register int temp;

 temp = 1;

 for(; e; e--) temp = temp * m;
 return temp;
}
```

В данном примере переменные **e**, **m** и **temp** объявлены как регистровые, поскольку они используются внутри циклов. Тот факт, что регистровые переменные оптимизированы по скорости, делает их идеально подходящими для использования в циклах. Обычно регистровые переменные применяются там, где к одной и той же переменной происходит очень много обращений. Это очень важно, поскольку регистровыми можно объявить сколько угодно переменных, но не все они будут обеспечивать одинаковую скорость доступа.

Количество регистровых переменных в каждом блоке оптимизируется по скорости и зависит как от операционной системы, так и от реализации языка C/C++. Беспокоиться о количестве регистровых переменных не стоит, поскольку компилятор автоматически преобразовывает "лишние" регистровые переменные в обычные. (Это обеспечивает машинонезависимость программ в рамках довольно широкого спектра процессоров.)

Обычно в регистрах центрального процессора одновременно могут располагаться, по крайней мере, две переменные типа **char** или **int**. Поскольку выбор операционных систем очень широк, универсальных рекомендаций по использованию регистровых переменных не существует. В каждом конкретном случае следует обращаться к документации компилятора.

В языке С адрес регистровой переменной с помощью оператора **&** вычислить невозможно (причины обсуждаются ниже). Это вполне логично, поскольку регистровые переменные хранятся, как правило, в процессоре, а его память не адресуется. Однако на язык C++ это ограничение не распространяется, хотя в этом случае вычисление адреса регистровой переменной может помешать оптимизации программы.

Несмотря на то что в настоящее время смысл спецификатора **register** существенно расширен по сравнению с традиционным, на практике он по-прежнему применяется в основном к целочисленным и символьным переменным. Таким образом, не следует ожидать от спецификатора **register** существенного ускорения работы программы, если он применяется к переменным других типов.

Инициализация переменных

При объявлении переменной ей можно присвоить начальное значение. Общий вид инициализации выглядит следующим образом.

тип имя_переменной = значение;

Рассмотрим несколько примеров.

```
char ch = 'a';
int first = 0;
float balance = 123.23;
```

Глобальные и локальные статические переменные инициализируются только при запуске программы. Локальные переменные (за исключением статических) инициализируются каждый раз при входе в блок, где они описаны. Неинициализированные локальные переменные имеют неопределенное значение, пока к ним не будет применен оператор присваивания. Неинициализированные глобальные переменные и локальные статические переменные автоматически устанавливаются равными нулю.

Константы

Константами (constants) называются фиксированные значения, которые программа не может изменить. Способ представления константы зависит от ее типа. Иногда константы также называют *литералами* (literal).

Символьные константы заключаются в одинарные кавычки. Например, символы **'a'** и **'%'** являются константами. В языках С и C++ предусмотрены расширенные символы, которые позволяют использовать другие языки, помимо английского. Их длина равна 16 бит. Для того чтобы определить расширенную символьную константу, перед символом следует поставить букву **L**. Рассмотрим пример.

```
wchar_t wc;
wc = L'A';
```

Здесь переменной **wc** присваивается расширенная символьная константа, эквивалентная букве А. Расширенные символы имеют тип **wchar_t**. В языке С этот тип определяется в заголовочном файле и не является встроенным типом, в отличие от языка C++, где он относится к элементарным типам.

Целочисленные константы считаются числами, не имеющими дробной части. Например, числа 10 и −100 являются целочисленными константами. Константы с плавающей точкой содержат дробную часть, которая отделяется десятичной точкой. Например, число 11.123 представляет собой константу с плавающей точкой. Кроме того, в языке C++ числа с плавающей точкой можно представлять с помощью научного формата.

В языке C предусмотрено два типа для представления чисел с плавающей точкой: **float** и **double**. Используя модификаторы, можно создать еще несколько вариантов основных типов. По умолчанию компилятор приводит числовые константы к наименьшему подходящему типу. Таким образом, если предположить, что целое число занимает 16 бит, то число 10 по умолчанию имеет тип **int**, а число 103000 — тип **long**. Несмотря на то что число 10 можно привести к символьному типу, компилятор не нарушит границы типов. Единственное исключение из этого правила представляет собой константа с плавающей точкой, которая по умолчанию имеет тип **double**.

В большинстве программ соглашения, принятые по умолчанию, являются вполне разумными. Однако, используя суффиксы, типы констант можно задавать явно. Если после константы с плавающей точкой поставить суффикс **F**, она будет иметь тип **float**. Если вместо буквы **F** поставить букву **L**, константа получит тип **long double**. Для целочисленных типов суффикс U означает **unsigned**, а суффикс **L** — **long**. Вот несколько примеров на эту тему.

Тип данных	*Примеры констант*
int	1 123 21000 -234
long int	35000L -34L
unsigned int	10000U 987U 40000U
float	123.23F 4.34e-3F
double	123.23 1.0 -0.9876324
long double	1001.2L

Шестнадцатеричные и восьмеричные константы

Иногда позиционные системы счисления по основанию 8 или 16 оказываются удобнее, чем обычная десятичная система. Позиционная система счисления по основанию 8 называется *восьмеричной* (octal). В ней используются цифры от 0 до 7. Числа в восьмеричной системе раскладываются по степеням числа 8. Система счисления с основанием 16 называется *шестнадцатеричной* (hexadecimal). В ней используются цифры от 0 до 9 и буквы от A до F, которые соответствуют числам 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 в десятичной системе равно 16. Поскольку эти системы счисления применяются очень часто, в языке C/C++ предусмотрены средства для представления шестнадцатеричных и восьмеричных констант. Для этого перед шестнадцатеричным числом указывается префикс **0x**. Восьмеричные константы начинаются с нуля. Рассмотрим несколько примеров.

```
int hex = 0x80; /* 128 в десятичной системе */
int oct = 012;  /* 10 в десятичной системе */
```

Строковые константы

В языке C/C++ есть еще один вид констант — строковые. *Строка* (string) — это последовательность символов, заключенная в двойные кавычки. Например, "пример строки" — это строка. Мы уже видели примеры использования строк, когда применяли функцию **printf()**. Несмотря на то что в языке C можно определять строковые константы, строго говоря, в нем нет отдельного типа данных для строк. (В то же время, в языке C++ существует стандартный класс **string**.)

Не следует путать символы и строки. Отдельная символьная константа заключается в одинарные кавычки, например `'a'`. Если заключить букву `a` в двойные кавычки, получим строку `"a"`, состоящую из одной буквы.

Управляющие символьные константы

Практически все символы можно вывести на печать, заключив их в одиночные кавычки. Однако некоторые символы, например, символ перехода на новую строку, невозможно ввести в строку с клавиатуры. Для этого в языке C/C++ предусмотрены специальные *управляющие символьные константы* (backslash character constants), указанные в табл. 2.2. Они называются *эскейп-последовательностями* (escape sequences). Для того чтобы гарантировать машинонезависимость, необходимо использовать не ASCII-коды, эквивалентные управляющим символьным константам, а именно эскейп-последовательности.

Таблица 2.2. Управляющие символьные константы

Код	*Значение*
`\b`	Пробел
`\f`	Прогон бумаги
`\n`	Новая строка
`\r`	Возврат каретки
`\t`	Горизонтальная табуляция
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\0`	Нуль
`\\`	Обратная косая черта
`\v`	Вертикальная табуляция
`\a`	Звуковой сигнал
`\?`	Знак вопроса
`\N`	Восьмеричная константа N
`\xN`	Шестнадцатеричная константа N

Программа, приведенная ниже, выполняет переход на новую строку, выводит символ табуляции, а затем печатает строку "Проверка вывода".

```
#include <stdio.h>

int main(void)
{
 printf("\n\tПроверка вывода.");

 return 0;
}
```

Операторы

В языке C/C++ предусмотрено большое количество операторов. В этих языках операторам уделяется гораздо больше внимания, чем в большинстве других языков. Операторы разделяются на четыре основные группы: *арифметические* (arithmetic), *сравнения* (relational), *логические* (logical) и *побитовые* (bitwise). Кроме того, для конкретных целей предусмотрено еще несколько специальных операторов.

Оператор присваивания

Оператор присваивания можно использовать в любом корректном выражении. В языке C/C++ (в отличие от многих языков программирования, включая Pascal, BASIC и FORTRAN) оператор присваивания не считается особенным. Общий вид оператора присваивания выглядит следующим образом.

имя_переменной = выражение;

Здесь выражение может состоять как из отдельной константы, так и комбинации сложных операторов. В качестве оператора присваивания в языке C/C++ используется знак равенства (отличие от языков Pascal и Modula-2, в которых оператор присваивания обозначается символами `:=`). *Цель* (target), или левая часть оператора присваивания, должна быть либо переменной, либо указателем, но не функцией или константой.

Часто в книгах о языке C/C++ и в сообщениях компилятора встречаются два термина: lvalue и rvalue. Термин *lvalue* означает любой объект, который может стоять в левой части оператора присваивания. С практической точки зрения термин lvalue относится к переменным. Термином *rvalue* называют выражение, стоящее в правой части оператора присваивания, точнее говоря, значение этого выражения.

Преобразования типов в операторе присваивания

Когда в выражении смешиваются переменные разных типов, необходимо выполнить *преобразование типов* (type conversion). Для оператора присваивания правило преобразования типов формулируется просто: значение правой части (rvalue) преобразовывается к типу левой части (lvalue). Рассмотрим конкретный пример.

```
int x;
char ch;
float f;

void func(void)
{
 ch = x; /* Строка 1 */
 x = f;  /* Строка 2 */
 f = ch; /* Строка 3 */
 f = x;  /* Строка 4 */
}
```

В строке 1 левый старший разряд целой переменной **x** отбрасывается, и переменной **ch** присваиваются 8 младших битов. Если значение переменной **x** изменяется от 0 до 255, то переменные **ch** и **x** будут эквивалентными. (Это относится лишь к тем компиляторам, в которых переменные типа **char** по умолчанию не имеют знака. Если тип **char** по умолчанию имеет знак, то переменные **ch** и **x** будут эквивалентными, только если переменная **x** изменяется от −128 до 127 либо переменная **ch** имеет тип **unsigned char** — *Прим. ред.*) В противном случае в переменной **ch** будут учтены лишь 8 младших бит переменной **x**. В строке 2 переменной **x** будет присвоена целая часть переменной **f**. В строке 3 восьмибитовое целое число, хранящееся в переменой **ch**, будет преобразовано в число с плавающей точкой. То же самое произойдет и в строке 4, только теперь в число с плавающей точкой будет преобразовано целочисленное значение переменной **x**.

В процессе преобразования типа **int** в тип **char** и типа **long int** в тип **int** старшие разряды отбрасываются. Во многих 16-битовых операционных системах это означает, что 8 бит будет потеряно. В 32-разрядных операционных системах при преобра-

зовании типа **int** в тип **char** будет потеряно 24 бит, а при преобразовании типа **int** в тип **short int** — 16 бит.

Правила преобразования типов суммированы в табл. 2.3. Помните, что преобразование типа **int** в тип **float**, типа **float** в тип **double** и так далее не увеличивает точности. Этот вид преобразований лишь изменяет способ представления числа. Кроме того, некоторые компиляторы при преобразовании типа **char** в тип **int** или **float** по умолчанию считают тип **char** не имеющим знака, независимо от значения переменной. Другие компиляторы преобразовывают значения символьных переменных, превышающие 127, в отрицательные числа. Итак, если вы хотите избежать проблем, для работы с символами следует использовать тип **char**, а для работы с целыми числами — тип **int**, **short int** или **signed char**.

Если вам необходимо выполнить преобразование типа, не указанное в табл. 2.3, преобразуйте его сначала в промежуточный тип, который есть в таблице, а затем — в результирующий тип. Например, чтобы преобразовать тип **double** в тип **int**, сначала следует преобразовать тип **double** в тип **float**, а затем тип **float** — в тип **int**. (Разумеется, это относится лишь к анализу возможных потерь точности. Любой компилятор совершенно свободно выполнит преобразование типа **double** в тип **int**, не требуя промежуточных преобразований. — *Прим. ред.*)

Таблица 2.3. Правила преобразования типов

Результирующий тип	*Тип выражения*	*Возможные потери*
`signed char`	`char`	Если значение > 125, результатом будет отрицательное число
`char`	`short int`	Старшие 8 бит
`char`	`int (16 бит)`	Старшие 8 бит
`char`	`int (32 бит)`	Старшие 24 бит
`char`	`long int`	Старшие 24 бит
`short int`	`int (16 бит)`	Нет
`short int`	`int (32 бит)`	Старшие 16 бит
`int (16 бит)`	`long int`	Старшие 16 бит
`int (32 бит)`	`long int`	Нет
`int`	`float`	Дробная часть и, возможно, что-то еще
`float`	`double`	Точность, результат округляется
`double`	`long double`	Точность, результат округляется

Множественные присваивания

В языке C/C++ разрешается присваивать одно и то же значение нескольким переменным одновременно. Например, во фрагменте программы, приведенном ниже, переменным **x**, **y** и **z** одновременно присваивается число 0.

```
x = y = z = 0;
```

Такой способ присваивания часто применяется в профессиональных программах.

Арифметические операторы

В табл. 2.4 приведен список арифметических операторов языка C/C++. Операторы **+**, **–**, ***** и **/** выполняются точно так же, как и в большинстве других языков программирования. Их можно применять практически к любым встроенным типам данных. Если оператор **/** применяется к целому числу или символу, дробная часть отбрасывается. Например, 5/2 равно 2.

Таблица 2.4. Арифметические операторы

Оператор	*Действие*
-	Вычитание, а также унарный минус
+	Сложение
*	Умножение
/	Деление
%	Деление по модулю
--	Декрементация
++	Инкрементация

Оператор деления по модулю %, как и в других языках программирования, возвращает остаток целочисленного деления. Однако этот оператор нельзя применять к числам с плавающей точкой. Рассмотрим пример, в котором используется оператор %.

```
int x, y;

x = 5;
y = 2;

printf("%d ", x/y); /* Выведет число 2 */
printf("%d ", x%y); /* Выведет число 1, остаток
                       целочисленного деления */

x = 1;
y = 2;

printf("%d %d", x/y, x%y); /* Выведет числа 0 1 */
```

Последний оператор программы выводит на экран число 0 и 1, поскольку результат целочисленного деления 1/2 равен 0, а остаток — 1.

Унарный минус умножает свой операнд на −1. Иными словами, унарный минус меняет знак операнда на противоположный.

Инкрементация и декрементация

В языке C/C++ есть два полезных оператора, которыми не обладают некоторые другие языки. Это операторы инкрементации и декрементации ++ и --. Оператор ++ добавляет 1 к своему операнду, а оператор -- вычитает ее. Таким образом, оператор

```
x = x+1;
```

эквивалентен оператору

```
++x;,
```

а оператор

```
x = x-1;
```

эквивалентен оператору

```
x--;
```

Операторы инкрементации и декрементации имеют две формы: префиксную и постфиксную. Например, фрагмент

```
x = x+1;
```

можно переписать так

```
++x;
```

или так

```
x++;
```

Однако между префиксной и постфиксной формами существует важное отличие, когда они используются внутри выражений. Если используется префиксная форма, операторы инкрементации и декрементации применяются к старому значению операнда, а если постфиксная — к новому. Например, в результате выполнения операторов

```
x=10;
y = ++x;
```

переменной **y** будет присвоено значение 11. Однако, если переписать этот фрагмент иначе:

```
x = 10;
y = x++;,
```

переменная **y** будет равна 10. В обоих случаях переменная **x** в результате станет равной 11. Разница заключается в том, когда это происходит.

Большинство компиляторов языка C/C++ создают для операторов инкрементации и декрементации очень быстрый и эффективный объектный код. Он выполняется намного быстрее, чем код, соответствующий оператору присваивания. По этой причине операторы инкрементации и декрементации следует применять всегда, когда это возможно.

Ниже приведены приоритеты арифметических операторов.

высший	**++ --**
	- (унарный минус)
	*** / %**
низший	**+ -**

Операторы, имеющие одинаковый приоритет, выполняются слева направо. Разумеется, для изменения порядка вычисления операторов можно применять скобки. В языке C/C++ скобки интерпретируются точно так же, как и во всех других языках программирования. Они позволяют присвоить некоторым операторам или группе операторов более высокий приоритет.

Операторы сравнения и логические операторы

В термине *оператор сравнения* слово "сравнение" относится к значениям операндов. В термине *логический оператор* слово "логический" относится к способу, которым устанавливаются эти отношения. Поскольку операторы сравнения и логические операторы тесно связаны друг с другом, мы рассмотрим их вместе.

В основе и операторов сравнения, и логических операторов лежат понятия "истина" и "ложь". В языке C истинным считается любое значение, не равное нулю. Ложное значение всегда равно 0. Выражения, использующие операторы сравнения и логические операторы, возвращают 0, если результат ложен, и 1, если результат истинен.

В языке C++ истинные и ложные значения различаются точно так же, но, помимо этого, предусмотрен особый тип данных **bool** и булевы константы **true** и **false**. В программах на языке C++ значение 0 автоматически преобразовывается в константу **false**, а ненулевое значение — в константу **true**. Справедливо и обратное утверждение: константа **true** преобразовывается в значение 1, а константа **false** — в число 0. Результатом выражений, в которых используются операторы сравнения и логические операторы, являются константы **true** и **false**. Однако, поскольку эти

константы автоматически преобразовываются в числа 1 или 0, различие между языками C и C++ в этом аспекте становится чисто академическим.

Операторы сравнения и логические операторы приведены в табл. 2.5.

Таблица 2.5. Операторы сравнения и логические операторы

Операторы сравнения	
Оператор	**Действие**
>	Больше
>=	Больше или равно
<	Меньше
<=	Меньше или равно
!=	Не равно
Логические операторы	
Оператор	**Действие**
&&	И
||	ИЛИ
!	НЕ

Рассмотрим таблицу истинности для логических операторов, используя значения 1 и 0.

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Операторы сравнения и логические операторы имеют более низкий приоритет, чем арифметические операторы. Таким образом, выражение `10 > 1 + 12` будет вычислено так, будто оно записано следующим образом: `10 > (1 + 12)`. Разумеется, в обоих случаях это выражение будет ложным.

В одном и том же выражении можно использовать несколько операторов.

```
10>5 && !(10<9) || 3<=4
```

В данном случае результатом будет истинное значение.

Несмотря на то что ни в языке C, ни в языке C++ нет оператора "исключительное ИЛИ" (XOR), его можно легко реализовать в виде функции, используя другие логические операторы. Результатом оператора XOR является истинное значение, только если один из операндов (но не оба) истинен. Программа, приведенная ниже, содержит функцию **`xor()`**, применяющую операцию исключающего ИЛИ к двум своим аргументам.

```
#include <stdio.h>

int xor(int a, int b);

int main(void)
{
 printf("%d", xor(1, 0));
 printf("%d", xor(1, 1));
 printf("%d", xor(0, 1));
 printf("%d", xor(0, 0));

 return 0;
}
```

```
/* Выполняет операцию исключающего ИЛИ,
 используя два аргумента. */
int xor(int a, int b)
{
 return (a || b) && !(a && b);
}
```

Ниже приведены приоритеты операторов сравнения и логических операторов.

высший	**!**
	> >= < <=
	== !=
	&&
низший	**||**

Как и для арифметических операторов, естественный порядок вычислений можно изменять с помощью скобок. Например, выражение

```
!0 && 0 || 0
```

является ложным. Однако, если перегруппировать его операнды с помощью скобок, результат станет противоположным.

```
!(0 && 0)) || 0
```

Помните: все выражения, использующие операторы сравнения и логические операторы, имеют либо истинное, либо ложное значение. Следовательно, фрагмент программы, приведенный ниже, верен. В результате его выполнения на экран будет выведено значение 1.

```
int x;

x = 100;
printf("%d", x>10);
```

Побитовые операторы

В отличие от многих других языков, язык C/C++ содержит полный набор побитовых операторов. Поскольку язык C был разработан в качестве замены языка ассемблера, он предусматривает многие операции низкого уровня, в частности, *побитовые операции* (bitwise operation), предназначенные для проверки, установки и сдвига битов, из которых состоят байты и машинные слова, образующие переменные типа **char** или **int**. Побитовые операции нельзя применять к переменным типа **float**, **double**, **long double**, **void**, **bool** и других, более сложных, типов. Побитовые операторы языка C/C++ перечислены в табл. 2.6. Эти операторы применяются к отдельным битам операндов.

Таблица 2.6. Побитовые операторы

Оператор	Действие
&	И
|	ИЛИ
^	Исключающее ИЛИ
~	Дополнение до единицы (НЕ)
>>	Сдвиг вправо
<<	Сдвиг влево

Побитовые операции И, ИЛИ и НЕ (дополнение до единицы) описываются теми же таблицами истинности, что и их логические аналоги, за исключением того, что теперь все сравнения проводятся поразрядно. Таблица истинности для оператора исключающего ИЛИ приведена ниже.

p	q	p^q
0	0	0
1	0	1
1	1	0
0	1	1

Как следует из этой таблицы, результат операции исключающего ИЛИ равен истинному значению тогда и только тогда, когда только один из операндов является истинным, в противном случае его значение является ложным.

Эти операторы часто применяются при разработке драйверов, например, программ, управляющих работой модема, диска или принтера, поскольку с помощью побитовых операций можно определить значения каждого конкретного бита, например, бита четности. (Бит четности подтверждает, что остальные биты не подвергались изменениям. Обычно это — старший бит в каждом байте.)

С помощью побитового оператора И можно сбросить значение бита, т.е. любой бит, равный нулю в каком-либо операнде, обнуляет значение соответствующего бита в результате. Например, приведенная ниже функция считывает символ из порта модема и обнуляет бит четности.

```
char get_char_from_modem(void)
{
 char ch;

 ch = read_modem(); /* Считывает символ из порта модема */
 return(ch & 127);
}
```

Четность обычно задается восьмым битом, который обнуляется в результате выполнения операции И. Вторым операндом этого оператора является байт, в котором все биты от 1 до 7 равны 1, а восьмой бит равен 0. Выражение **ch&127** означает попарное применение операции И ко всем битам, составляющим значение переменной **ch**, и битам числа 127. В результате восьмой бит переменной **ch** становится равным 0. Предположим, что в переменная **ch** содержит символ **"A"** и бит четности.

```
Бит четности
 ↓
 1 1 0 0 0 0 0 1 Переменная ch, содержащая символ "a" и бит четности
 0 1 1 1 1 1 1 1 Двоичное представление числа 127
& _____________  Побитовый оператор И
 0 1 0 0 0 0 0 1 Символ "A" без бита четности
```

Побитовая операция ИЛИ, в отличие от операции И, используется для установки значения бита. Любой бит, равный 1 в каком-либо операнде, присваивает 1 соответствующему биту в результате. Посмотрим, например, чему равно выражение 128 | 3.

```
 1 0 0 0 0 0 0 0 Двоичное представление числа 128
 0 0 0 0 0 1 1 1 Двоичное представление числа 3
 |_____________  Побитовый оператор ИЛИ
 1 0 0 0 0 1 1 1 Результат
```

Операция исключающего ИЛИ, обычно сокращенно называемая XOR, устанавливает значение бита тогда и только тогда, когда оба сравниваемых бита имеют разные значения. Например, выражение 127^120 вычисляется следующим образом.

```
0 1 1 1 1 1 1 1 Двоичное представление числа 127
0 1 1 1 0 0 0 0 Двоичное представление числа 120
^_____________  Побитовый оператор исключающего ИЛИ
0 0 0 0 1 1 1 1 Результат
```

Помните, что результатом работы операторов сравнения и логических операторов всегда являются истинные или ложные значения, в то время как аналогичные побитовые операторы могут порождать любые целые числа. Иными словами, результатом побитовых операций являются любые числа, а не только 0 или 1, как при использовании логических операторов.

Операторы сдвига битов ">>" и "<<" смещают все биты влево или вправо на указанное количество разрядов. Общий вид операторов побитового сдвига вправо выглядит следующим образом.

значение >> количество разрядов

Оператор побитового сдвига влево выглядит так.

значение << количество разрядов

Как только сдвигаемые биты достигают края, с противоположного конца появляются нули. (При сдвиге вправо в отрицательном целом числе признак знака сохраняется.) Помните, что сдвиг не является кольцевым, т.е. если биты исчезают на одном краю числа, они не появятся на другом. Биты, вышедшие за пределы числа, считаются потерянными.

Операции побитового сдвига могут быть очень полезны при декодировании информации, поступающей с внешнего устройства, например модема. Кроме того, побитовые операторы сдвига можно использовать для быстрого умножения или деления целых чисел. С помощью сдвига вправо можно эффективно поделить число на 2, а с помощью сдвига влево — умножить на 2, как показано в табл. 2.7.

Таблица 2.7. Умножение и деление с помощью побитовых сдвигов

unsigned char x	*Двоичное представление*	*Значение*
x=7;	0 0 0 0 0 1 1 1	7
x=x<<1;	0 0 0 0 1 1 1 0	14
x=x<<3	0 1 1 1 0 0 0 0	112
x=x<<2	1 1 0 0 0 0 0 0	192
x=x>>1	0 1 1 0 0 0 0 0	96
x=x>>2	0 0 0 1 1 0 0 0	24

Примечание. Каждый сдвиг битов влево означает умножение на 2. После выполнения оператора **x<<2** часть информации пропадет, поскольку биты выходят за границы числа. Каждый сдвиг битов вправо означает деление на 2. Обратите внимание на то, что последующие деления числа не возвращают потерянные биты.

Проиллюстрируем операции побитового сдвига следующей программой.

```
/* Пример побитового сдвига. */
#include <stdio.h>

int main(void)
{
 unsigned int i;
 int j;

 i = 1;

 /* Сдвиги влево */
 for(j=0; j<4; j++) {
```

```
 i = i << 1; /* Сдвиг влево на 1 бит означает умножение на 2*/
 printf("Сдвиг влево %d: %d\n", j, i);
 }

 /* Сдвиги вправо */
 for(j=0; j<4; j++) {
 i = i >> 1; /* Сдвиг вправо на 1 бит означает деление на 2*/
 printf("Сдвиг вправо %d: %d\n", j, i);
 }

 return 0;
}
```

Оператор дополнения до единицы "**~**" инвертирует каждый бит операнда. Это значит, что каждая единица станет нулем, и наоборот.

Побитовые операторы часто используются в шифровальных программах. Если вам необходимо сделать файл нечитаемым, достаточно проделать над ним несколько побитовых манипуляций. Проще всего инвертировать каждый бит в байтах, как показано ниже.

```
Исходный байт             0 0 1 0 1 1 0 0
После первого отрицания 1 1 0 1 0 0 1 1
После второго отрицания 0 0 1 0 1 1 0 0
```

Обратите внимание на то, что двойное отрицание всегда приводит к исходному значению. Следовательно, первое дополнение до единицы зашифрует байт, а второе — восстановит его.

Для кодирования символов можно применять функцию, приведенную ниже.

```
/* Простая шифровальная функция. */
char encode(char ch)
{
 return(~ch); /* Дополнение до единицы */
}
```

Разумеется, файл, закодированный с помощью функции **encode()**, очень легко расшифровать!

Тернарный оператор

Язык C/C++ содержит очень мощный и удобный оператор, заменяющий некоторые операторы вида **if-then-else**. Тернарный оператор "**?**" имеет следующий вид.

Выражение_1 ? *Выражение_2* : *Выражение_3*;

Оператор "**?**" выполняется следующим образом: сначала вычисляется *Выражение_1*. Если оно истинно, вычисляется *Выражение_2*, и его значение становится результатом всего оператора. Если *Выражение_1* ложно, вычисляется *Выражение_3*, и его значение становится результатом оператора. Рассмотрим фрагмент программы.

```
x = 10;

y = x>9 ? 100 : 200;
```

Здесь переменной **y** присваивается значение 100. Если бы значение переменной x было меньше 9, то переменная **y** стала бы равно 200. Тот же самый код можно записать с помощью оператора **if-else**.

```
x = 10;
```

```
if(x>9) y = 100;
else y = 200;
```

Более подробно оператор "?" обсуждается в главе 3, в которой рассматриваются другие условные операторы.

Операторы взятия адреса и разыменования

Указатель[1] (pointer value) — это переменная, объявленная особым образом, в которой хранится адрес объекта определенного типа. Можно выделить три основные области применения указателей в языке C/C++. Во-первых, с их помощью легко индексировать элементы массивов. Во-вторых, они позволяют функциям модифицировать свои параметры. И, в-третьих, на основе указателей можно создавать связанные списки и другие динамические структуры данных. Подробнее указатели описываются в главе 5, а пока мы изучим два оператора, позволяющих ими манипулировать.

Первый из них — унарный *оператор взятия адреса* **&**, возвращающий адрес своего операнда. (Унарным называется оператор, имеющий только один операнд.) Например, оператор

```
m = &count;
```

помещает в переменную **m** адрес переменной **count**. Этот адрес относится к ячейке памяти, в которой находится переменная **count**, и никак не связан с ее значением. Выражение **&count** можно перевести как "адрес переменной **count**". Таким образом, оператор присваивания, приведенный выше, означает: "Присвоить переменной **m** адрес переменной **count**".

Чтобы лучше понять, что при этом происходит, допустим, что переменная **count** расположена в ячейке с адресом 2000, а ее значение равно 100. Тогда в предыдущем примере переменной **m** будет присвоено число 2000.

Второй оператор — *оператор разыменования указателя* *, являющийся противоположностью оператора **&**. Этот унарный оператор возвращает значение объекта, расположенного по указанному адресу. Например, если переменная **m** содержит адрес переменной **cout**, то оператор

```
q = *m;
```

присвоит значение переменной **count** переменной **q**. Теперь значение переменной **q** будет равно 100, поскольку именно это число записано в ячейке, в которой хранится переменная **m**. Оператор разыменования указателя * можно перевести как "значение, хранящееся по адресу". Фрагмент программы, показанный выше, читается следующим образом: "присвоить переменной **q** значение, хранящееся по адресу **m**". К сожалению, символ оператора разыменования указателя совпадает с символом операции умножения, а символ оператора взятия адреса — с символом оператора побитового И. Эти операторы никак не связаны друг с другом. Приоритет операторов разыменования указателя * и взятия адреса **&** выше, чем приоритет любого арифметического оператора, за исключением унарного минуса, имеющего такой же приоритет.

В объявлении между именем типа и именем указателя ставится символ *. Он сообщает компилятору, что объявляемая переменная является указателем на переменную заданного типа. Например, объявление указателя на переменную типа **char** записывается следующим образом:

```
char *ch;
```

---

[1] В оригинале автор использует несколько иную терминологию, называя указателем адрес переменной, а то, что мы называем указателем, — переменной-указателем. Как нам кажется, это не вполне соответствует как нормам литературного языка, так и традициям, установившимся в русскоязычной литературе по программированию. — *Прим. ред.*

Обратите внимание на то, что переменная **ch** в данном фрагменте программы является указателем на переменную типа **char**, а не символьной переменной. Именно в этом заключается их принципиальное различие. Тип переменной, на которую ссылается указатель (в данном случае тип **char**), называется *базовым типом указателя*. Сам указатель представляет собой переменную, в которой хранится адрес объекта базового типа. Таким образом, указатель на символьную переменную (как и любой другой указатель) должен иметь достаточный размер, чтобы хранить любой адрес в соответствии с архитектурой компьютера. Однако, как правило, указатели ссылаются только на объекты своего базового типа.

В одном объявлении можно смешивать и указатели, и обычные переменные. Например, оператор

```
int x, *y, count;
```

объявляет целочисленные переменные **x** и **count** вместе с указателем **y**, ссылающимся на целочисленные переменные.

В следующей программе операторы * и & используются для записи значения 10 в переменную **target**. В результате на экран выводится число 10.

```
#include <stdio.h>

int main(void)
{
 int target, source;
 int *m;

 source = 10;
 m = &source;
 target = *m;

 printf("%d", target);

 return 0;
}
```

Статический оператор sizeof

Статический унарный оператор **sizeof** вычисляет длину операнда в байтах. Операндом может быть как отдельная переменная, так и имя типа, заключенное в скобки. Предположим, что размер типа **int** равен 4 байт, а типа **double** — 8 байт. В этом случае следующий фрагмент программы выведет на экран числа **8 4**.

```
double f;

printf("%d ", sizeof f);
printf("%d ", sizeof(int));
```

Следует помнить, что для вычисления размера типа его имя должно быть заключено в скобки. Имя переменной в скобки заключать не обязательно, хотя и вреда от этого не будет.

С помощью оператора переопределения типа **typedef** в языке C/C++ выделен специальный тип **size_t**, отдаленно напоминающий целое число без знака. С формальной точки зрения результат оператора **sizeof** имеет тип **size_t**, но на практике его можно считать целым числом без знака.

Оператор **sizeof** позволяет создавать машинонезависимые программы, в которых автоматически учитываются размеры встроенных типов. В качестве примера рассмотрим базу данных, каждая запись которой состоит из шести целых чисел. Если такую базу необходимо реализовать на разных компьютерах, нельзя надеяться, что размер целочис-

ленного типа везде будет одинаков. Фактическую длину типа следует определить с помощью оператора **sizeof**. Таким образом, программа должна иметь примерно такой вид.

```
/* Запись 6 целых чисел в файл. */
void put_rec(int rec[6], FILE *fp)
{
 int len;

 len = fwrite(rec, sizeof(int)*6, 1, fp);
 if(len != 1) printf("Ошибка при записи");
}
```

Функция **put_rec()** правильно компилируется и выполняется в любой среде, в том числе на 16- и 32-разрядных компьютерах.

В заключение отметим, что оператор **sizeof** выполняется на этапе компиляции, и его результат в программе рассматривается как константа.

Оператор последовательного вычисления

Оператор последовательного вычисления (operator comma) связывает в одно целое несколько выражений. Символом этого оператора является запятая. Подразумевается, что левая часть оператора последовательного вычисления всегда имеет тип **void**. Это означает, что значение выражения, стоящего в его правой части, становится значением всего выражения, разделенного запятыми. Например, оператор

```
x = (y=3, y+1);
```

сначала присваивает переменной **y** значение 3, а затем присваивает переменной **x** значение 4. Скобки здесь необходимы, поскольку приоритет оператора последовательного вычисления меньше, чем приоритет оператора присваивания.

По существу, выполнение оператора последовательного вычисления сводится к выполнению нескольких операторов подряд. Если этот оператор стоит в правой части оператора присваивания, то его результатом всегда будет результат выражения, стоящего последним в списке.

Оператор последовательного вычисления можно сравнить с союзом "и" в выражении "сделай это и это и это".

Оператор доступа к члену структуры и ссылки на член структуры

В языке C символы "**.**" и "**->**" обозначают операторы доступа к элементам структур и объединений. *Структуры* и *объединения* являются составными типами данных (иногда их еще называют *агрегированными*), в которых одно имя связывает несколько объектов. (см. главу 7). Оператор доступа к члену структуры (точка) используется для непосредственного обращения к элементу структуры или объединения. Оператор ссылки на член структуры используется для обращения к члену структуры или объединения через указатель. Рассмотрим фрагмент программы.

```
struct employee
{
 char name[80];
 int age;
 float wage;
} emp;

struct employee *p = &emp; /* Записать адрес структуры emp
                              в указатель p */
```

Для того чтобы присвоить члену **wage** структуры **emp** значение 123.33, нужно выполнить следующий оператор

```
emp.wage = 123.23;
```

То же самое можно сделать, использовав указатель на структуру **emp**.

```
p->wage = 123.23;
```

Операторы "[]" и "()"

Круглые скобки — это оператор, повышающий приоритет операций, заключенных внутри. Квадратные скобки используются для индексации массива (массивы подробно обсуждаются в главе 4). Если в программе определен массив, то значение выражения, заключенного в квадратные скобки, равно индексу элемента массива. Рассмотрим демонстрационную программу.

```
#include <stdio.h>
char s[80];

int main(void)
{
 s[3] = 'X';
 printf("%c", s[3]);

 return 0;
}
```

Здесь символ **'X'** сначала присваивается четвертому элементу массива (в языке C элементы всех массивов нумеруются, начиная с нуля), а затем выводится на экран.

Приоритеты операторов

Приоритеты всех операторов указаны в табл. 2.8. Обратите внимание на то, что все операторы, за исключением унарных и тернарного, выполняются слева направо. Унарные операторы "*", "&" и "–" выполняются справа налево.

Таблица 2.8. Приоритеты операторов в языке C

Высший	() [] ->
	! ~ ++ --+- (тип) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
Высший	?:
	= += -= *= /= и т.д.
Низший	,

На заметку *В языке C++ определены еще несколько дополнительных операторов, которые будут рассмотрены в части II.*

Выражения

Выражения состоят из операторов, констант, функций и переменных. В языке C/C++ выражением считается любая допустимая комбинация этих элементов. Поскольку большинство выражений в языке C/C++ напоминают алгебраические, часто их таковыми и считают. Однако при этом следует учитывать специфические особенности языков C и C++.

Порядок вычислений

Ни в языке C, ни в языке C++ порядок вычисления подвыражений, входящих в выражения, не определен. Компилятор может сам перестраивать выражения, стремясь к созданию оптимального объектного кода. Это означает, что программист не должен полагаться на определенный порядок вычисления подвыражений. Скажем, при вычислении выражения

```
x = f1() + f2();
```

нет никаких гарантий, что функция **f1()** будет вызвана раньше функции **f2()**.

Преобразование типов в выражениях

Если в выражение входят константы и переменные разных типов, они преобразуются к одному типу. Компилятор преобразует все операнды в тип "наибольшего" операнда. Этот процесс называется *расширением типов* (type promotion). Во-первых, все переменные типов **char** и **short int** автоматически преобразуются в тип **int**. Этот процесс называется *целочисленным расширением* (integral promotion). Во-вторых, все преобразования выполняются одно за другим, следуя приведенному ниже алгоритму.

```
ЕСЛИ операнд имеет тип long double,
ТО второй операнд преобразуется в тип long double
ИНАЧЕ, ЕСЛИ операнд имеет тип double,
ТО второй операнд преобразуется в тип double
ИНАЧЕ, ЕСЛИ операнд имеет тип float,
ТО второй операнд преобразуется в тип float
ИНАЧЕ, ЕСЛИ операнд имеет тип unsigned long,
ТО второй операнд преобразуется в тип unsigned long
ИНАЧЕ, ЕСЛИ операнд имеет тип long,
ТО второй операнд преобразуется в тип long
ИНАЧЕ, ЕСЛИ операнд имеет тип unsigned int,
ТО второй операнд преобразуется в тип unsigned int
```

Кроме того, есть одно правило: если один из операторов имеет тип **long**, а второй — **unsigned int**, и значение переменной типа **unsigned int** невозможно представить с помощью типа **long**, то оба операнда преобразуются в тип **unsigned long**.

После выполнения преобразований тип обоих операндов одинаков и совпадает с типом результата выражения.

Рассмотрим пример преобразования типов.[2]

[2] Предполагается, что переменная **result** описана в другом месте программы. — *Прим. ред.*

```
char ch;
int i;
float f;
double d;
r e s u l t = ( ch / i ) + ( f * d) - ( f + i )
                 |   |       |  |      |   |
               int   |    double |      | float
                 |___|       |__|      |___|
                   |          |         |
                  int       double    float
                   |_____________|       |
                         |               |
                         |_______________|
                                 |
                              double
```

Здесь символ **ch** сначала преобразуется в целое число. Затем результат операции **ch/i** преобразуется в тип **double**, поскольку именно этот тип имеет результат выражения **f*g**. После этого результат операции **f+i** преобразуется в тип переменной **f**, т.е. в тип **float**. Окончательный результат имеет тип **double**.

Приведение типов

Используя *приведение типов* (cast), можно принудительно задать тип выражения. Приведение типов имеет следующий вид.

(тип) выражение

Здесь *тип* — это любой допустимый тип данных. Например, в приведенном ниже фрагменте программы результат выражения **x/2** приводится к типу **float**.

```
(float) x/2
```

С формальной точки зрения приведение типов является унарным оператором, и его приоритет совпадает с приоритетами всех других унарных операторов.

Несмотря на то что приведение типов не слишком широко используется в программировании, иногда этот оператор оказывается очень полезным. Предположим, что нам необходимо использовать целочисленную переменную в качестве счетчика цикла, а в вычислениях необходимо учитывать дробные части. В этом случае приведение типов позволит нам сохранить точность вычислений.

```
#include <stdio.h>

int main(void) /* Выводит на печать число i и i/2
                  с учетом дробной части */
{
 int i;

 for(i=1; i<=100; ++i)
 printf("%d / 2 =: %f\n", i, (float) i /2);

 return 0;
}
```

Без оператора приведения типа **(float)** деление было бы целочисленным и дробная часть была бы потеряна.

На заметку *В языке C++ предусмотрены еще несколько операторов приввдения типов, в частности* ***const_cast*** *и* ***static_cast****. Они обсуждаются в части II.*

Пробелы и круглые скобки

Пробелы и символы табуляции облегчают чтение программ. Например, два следующих оператора совершенно одинаковы.

```
x=10/y~(127/x);
x = 10 / y ~(127/x);
```

Лишние скобки не приводят к ошибкам и не замедляют выполнение программ. Однако они позволяют прояснить точный порядок вычислений. Как вы полагаете, какой из операторов легче понять?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Составные операторы присваивания

В языке C/C++ существуют составные формы оператора присваивания, представляющие собой комбинации оператора присваивания и арифметических операторов. Например, оператор

```
x = x+10;
```

можно переписать в виде

```
x += 10;
```

Оператор "**+=**" сообщает компилятору, что к старому значению переменной **x** следует прибавить число 10.

Оператор присваивания образует составные формы практически со всеми бинарными операторами. Итак, выражение вида

переменная = переменная оператор выражение

можно переписать следующим образом:

переменная оператор= выражение

Например, выражение

```
x = x-100;
```

можно переписать как

```
x -= 100;
```

Составные формы присваивания широко используются в профессиональных программах на языке C/C++. Часто их называют *сокращенными операторами присваивания* (shorthand assignments), поскольку они более лаконичны.

Полный справочник по

Глава 3

Операторы

Оператор — это часть программы, которую можно выполнить отдельно.[3] Иными словами, оператор определяет некое действие. Операторы языка C и C++ разделяются на следующие категории.

- Условные операторы
- Операторы цикла
- Операторы перехода
- Метки
- Операторы-выражения
- Блоки

К *условным* (conditional) относятся операторы **if** и **switch**. (Условные операторы иногда называют *операторами ветвления* (selection statement).) *Операторы цикла* (iteration statements) обозначаются ключевыми словами **while**, **for** и **do while**. Группа операторов перехода состоит из операторов **break**, **continue**, **goto** и **return**. Метками служат операторы **case**, **default** (они рассматриваются в разделе "Оператор switch") и собственно метки, которые описываются в разделе "Оператор goto". Операторы-выражения — это операторы, состоящие из допустимых выражений. Блок представляет собой фрагмент текста программы, заключенный в фигурные скобки. Иногда блоки называют *составными операторами* (compound statements).

На заметку

В языке C++ есть еще два оператора: блок **try** *(для обработки исключительных ситуаций) и оператор объявления (declaration statement). Они обсуждаются в части II.*

Поскольку во многих операторах результат вычисления зависит от истинности или ложности некоторых проверок, начнем с понятий "истина" и "ложь".

Истинные и ложные значения в языках C и C++

При выполнении многих операторов языка C/C++ вычисляются значения условных выражений, которые имеют истинные или ложные значения. В языке C истинным считается любое ненулевое значение, в том числе отрицательное. Ложное значение всегда равно нулю. Такое представление истинных и ложных значений позволяет создавать чрезвычайно эффективные программы.

В языке C++ полностью поддерживается описанная выше концепция истинных и ложных значений. Наряду с этим в языке C++ используется булев тип данных с именем **bool**, который предусматривает только два значения: **true** и **false**. Как указано в главе 2, в языке C++ число 0 автоматически преобразовывается в значение **false**, а любое ненулевое значение — в значение **true**. Справедливо и обратное утверждение: значение **false** преобразуется в 0, а **true** — в 1. С формальной точки зрения условное выражение, входящее в условный оператор, имеет тип **bool**. Однако, поскольку любое ненулевое значение преобразуется в значение **true**, а число 0 — в значение **false**, между языками C и C++ в этом отношении нет никакой разницы.

---

[3] В книгах по языку C++ слово "statement" иногда переводится как "инструкция". В качестве аргумента приводят утверждение, что слово "оператор" в языке C++ относится к символу операции, который собственно и называется английским словом "operator". Не желая оспаривать эту точку зрения, заметим только, что традиционно любая строка программы в русскоязычной литературе по программированию называется оператором. Аналогично, пытаясь разрешить неоднозначность "строка" — "line" и "строка" — "string", мы будем вынуждены придумать название для символьной строки, например стринг. И так можно продолжать до бесконечности! Нам кажется, что разумнее придерживаться старых добрых правил и руководствоваться контекстом, поскольку перепутать строку программы с символом операции довольно трудно. — *Прим. ред.*

На заметку *В стандарте C99 предусмотрен булев тип* **_Bool**, *однако он несовместим с языком C++. Способы достижения совместимости между типами* **_Bool** *из языка C и* **bool** *из языка C++ обсуждаются во второй части книги.*

Условные операторы

В языке C/C++ предусмотрены два условных оператора: **if** и **switch**. Кроме того, в некоторых ситуациях в качестве альтернативы условному оператору **if** можно применять тернарный оператор "**?**".

Оператор if

Оператор **if** имеет следующий вид:

```
if (выражение) оператор;
else оператор;
```

Здесь *оператор* может состоять из одного или нескольких операторов или отсутствовать вовсе (пустой оператор). Раздел **else** является необязательным.

Если *выражение* истинно (т.е. не равно нулю), выполняется оператор или блок, указанный в разделе **if**, в противном случае выполняется оператор или блок, предусмотренный в разделе **else**. Операторы, указанные в разделах **if** или **else**, являются взаимоисключающими.

В языке C результатом условного выражения является *скаляр*, т.е. целое число, символ, указатель или число с плавающей точкой. В языке C++ к этому набору типов добавляется тип **bool**. Число с плавающей точкой редко применяется в качестве результата условного выражения, входящего в условный оператор, поскольку значительно замедляет выполнение программы. (Это объясняется тем, что операции над числами с плавающей точкой выполняются медленнее, чем над целыми числами или символами.)

Проиллюстрируем применение оператора **if** с помощью программы, которая представляет собой упрощенную версию игры "угадай волшебное число". Если игрок выбрал задуманное число, программа выводит на экран сообщение ****Верно****. "Волшебное число" генерируется с помощью стандартного датчика псевдослучайных чисел **rand()**, возвращающего произвольное число, лежащее в диапазоне от 0 до **RAND_MAX**. Как правило, это число не превышает 32767. Функция **rand()** объявлена в заголовочном файле **stdlib()**.

```
/* Волшебное число. Вариант #1. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* Волшебное число */
  int guess; /* Предположение игрока */

  magic = rand(); /* Генерируем волшебное число */

  printf("Угадайте волшебное число: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Верно **");

  return 0;
}
```

Следующая версия программы иллюстрирует применение оператора **else** в случае, если игрок ошибся.

```
/* Волшебное число. Вариант #2. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* Волшебное число */
  int guess; /* Предположение игрока */

  magic = rand(); /* Генерируем волшебное число */

  printf("Угадай волшебное число: ");
  scanf("%d", &guess);

  if(guess == magic) printf("*** Верно ***");
  else printf("Неверно");

  return 0;
}
```

Вложенные операторы if

Вложенным (nested) называется оператор **if**, который находится внутри другого оператора **if** или **else**. Вложенные операторы **if** встречаются довольно часто. Во вложенном условном операторе раздел **else** всегда связан с ближайшим оператором **if**, находящимся с ним в одном блоке и не связанным с другим оператором **else**. Рассмотрим пример.

```
if(i)
{
  if(j) оператор1;
  if(k) оператор2; /* данный if */
  else  оператор3; /* связан с данным оператором else */
}
else оператор 4; /* связан с оператором if(i) */
```

Последний раздел **else** связан не с оператором **if(j)**, который находится в другом блоке, а с оператором **if(i)**. Внутренний раздел **else** связан с оператором **if(k)**, потому что этот оператор **if** является ближайшим

Язык C допускает до 15 уровней вложенности условных операторов. На практике многие компиляторы предусматривают намного большую глубину. Гораздо важнее, что язык C++ допускает до 256 уровней вложения. Однако на практике глубоко вложенные условные операторы используются крайне редко, поскольку это значительно усложняет логику программы.

Используем вложенные операторы **if** для модификации нашей программы.

```
/* Волшебное число. Вариант #3. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* Волшебное число */
  int guess; /* Предположение игрока */
```

```
  magic = rand(); /* Генерируем волшебное число */

  printf("Угадайте волшебное число: ");
  scanf("%d", &guess);

  if (guess == magic) {
    printf("** Верно **");
    printf(" %d — волшебное число\n", magic);
  }
  else {
    printf("Неверно, ");
    if(guess > magic) printf("Слишком мало\n");
    else printf("Слишком много\n");
  }

  return 0;
}
```

Цепочка операторов if-then-else

В программах на языке C/C++ часто используется конструкция, которая называется *цепочка* **if-then-else** (if-then-alse ladder), или *лестница* **if-then-else** (if-then-else staircase). Общий вид этой конструкции выглядит так.

```
if(выражение) оператор;
else
  if (выражение) оператор;
  else
    if(выражение) оператор;




    else оператор;
```

Эти условия вычисляются сверху вниз. Как только значение условного выражения становится истинным, выполняется связанный с ним оператор, и оставшаяся часть конструкции игнорируется. Если все условные выражения оказались ложными, выполняется оператор, указанный в последнем разделе **else**. Если этого раздела нет, то не выполняется ни один оператор.

Поскольку по мере увеличения глубины вложенности количество отступов в строке возрастает, лестницу **if-then-else** часто записывают иначе

```
if(выражение)
  оператор;
else if (выражение)
  оператор;
else if (выражение)
  оператор;


    .
else
  оператор;
```

Используя цепочку операторов **if-then-else**, программу для угадывания "волшебного числа" можно переписать следующим образом.

```
/* Волшебное число. Вариант #4. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* Волшебное число */
  int guess; /* Предположение пользователя */

  magic = rand(); /* Генерируем волшебное число */

  printf("Угадайте волшебное число: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Верно ** ");
    printf("%d - волшебное число", magic);
  }
  else if(guess > magic)
    printf("Неверно, слишком много");
  else printf("Неверно, слишком мало");

  return 0;
}
```

Тернарная альтернатива

Вместо операторов **if-else** можно использовать тернарный оператор "**?**". Общий вид заменяемых операторов **if-else** выглядит следующим образом.

if (*условие*) *выражение*;
else *выражение*;

Однако в данном случае с операторами **if** и **else** связаны отдельные выражения, а не операторы.

Оператор "**?**" называется *тернарным*, поскольку имеет три операнда. Его общий вид таков.

Выражение1? Выражение2: Выражение3

Обратите внимание на использование и местоположение двоеточия.

Оператор "**?**" выполняется следующим образом. Сначала вычисляется *Выражение1*. Если оно является истинным, вычисляется *Выражение2*, и его значение становится значением всего тернарного оператора. Если *Выражение1* является ложным, вычисляется *Выражение3*, и результатом выполнения тернарного оператора считается именно его значение. Рассмотрим пример.

```
x = 10;
y = x>9 ? 100 : 200;
```

В данном случае переменной **y** присваивается значение 100. Если бы переменная **x** была меньше 9, то переменная **y** получила бы значение 200 Этот код можно переписать с помощью операторов **if-else**.

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

В следующем примере оператор "?" используется для "возведения в квадрат с сохранением знака" числа, введенного пользователем. Эта программа учитывает знак исходного числа следующим образом: например, 10 в квадрате равно 100, а −10 в квадрате равно −100.

```
#include <stdio.h>

int main(void)
{
  int isqrd, i;

  printf("Введите число: ");
  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);

  printf("%d в квадрате равно %d", i, isqrd);

  return 0;
}
```

Тернарный оператор можно использовать вместо конструкции **if-else** не только для присвоения значений. Как известно, все функции возвращают какое-либо значение (кроме функций, возвращающих значение типа **void**). Следовательно, вместо выражений в операторе "?" можно использовать вызовы функций. Если в операторе "?" встречается имя функции, она вызывается, а ее результат используется вместо значения соответствующего выражения. Это означает, что, используя вызовы функций в качестве операндов тернарного оператора, можно выполнить одну или несколько функций сразу. Рассмотрим пример.

```
#include <stdio.h>

int f1(int n);
int f2(void);

int main(void)
{
  int t;

  printf("Введите число: ");
  scanf("%d", &t);

  /* Вывод соответствующего сообщения */
  t ? f1(t) + f2() : printf("Введен нуль.\n");

  return 0;
}

int f1(int n)
{
  printf("%d ", n);
  return 0;
}

int f2(void)
{
  printf("введено.\n");
  return 0;
}
```

Если в программу введен нуль, вызывается функция **printf()**, и на экране появляется сообщение "Введен нуль". Если введено любое другое число, выполняются обе функции **f1()** и **f2()**. Обратите внимание на то, что результат оператора "**?**" в данной программе игнорируется и не присваивается ни одной переменной.

Учтите, компилятор языка C++, стремясь оптимизировать объектный код, может произвольно менять порядок вычисления выражений. В таком случае очередность вызовов функций **f1()** и **f2()** в операторе "**?**" остается неизвестной.

Используя тернарный оператор, можно еще раз переписать программу для угадывания "волшебного числа".

```
/* Волшебное число. Программа #5. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* Волшебное число */
  int guess; /* Предположение игрока */

  magic = rand(); /* Генерируем волшебное число */

  printf("Угадайте волшебное число: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Верно ** ");
    printf("%d - волшебное число", magic);
  }
  else
    guess > magic ? printf("Много") : printf("Мало");

  return 0;
}
```

В этой программе оператор "**?**" в зависимости от результата проверки выводит на экран соответствующие сообщения.

Условное выражение

Иногда начинающих программистов на языке C/C++ приводит в смятение тот факт, что в качестве условных выражений в операторах **if** или "**?**" можно использовать любые допустимые выражения. Иначе говоря, условное выражение не обязано состоять из операторов сравнения или логических операторов, как в языках BASIC и Pascal. Значением выражения может быть любое число, которое в зависимости от своего значения интерпретируется как "истина" или "ложь". Например, приведенный ниже фрагмент программы считывает с клавиатуры два целых числа и выводит на экран их частное.

```
/* Деление первого числа на второе. */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Введите два числа: ");
  scanf("%d%d", &a, &b);
```

```
    if(b) printf("%d\n", a/b);
    else printf("Делить на нуль нельзя.\n");

    return 0;
}
```

Если число **b** равно нулю, то условие оператора **if** является ложным, и выполняется оператор **else**. В противном случае условие является истинным, и выполняется деление.

Оператор **if** можно было бы записать иначе.

```
if(b != 0) printf("%d\n", a/b);
```

Однако следует заметить, что последняя запись избыточна, может снизить эффективность вычислений и поэтому считается признаком плохого стиля программирования. Для проверки условия оператора **if** достаточно значения самой переменной **b** и нет никакой необходимости сравнивать ее с нулем.

Оператор switch

В языке C/C++ предусмотрен оператор многовариантного ветвления **switch**, который последовательно сравнивает значение выражения со списком целых чисел или символьных констант. Если обнаруживается совпадение, выполняется оператор, связанный с соответствующей константой. Оператор **switch** имеет следующий вид.

```
switch (выражение)
{
  case константа1:
    последовательность операторов
    break;
  case константа2:
    последовательность операторов
    break;
  case константа3:
    последовательность операторов
    break;
  .
  .
  .
  default:
    последовательность операторов
}
```

Значением *выражения* должен быть символ или целое число. Например, выражения, результатом которых является число с плавающей точкой, не допускаются. Значение выражения последовательно сравнивается с константами, указанными в операторах **case**. Если обнаруживается совпадение, выполняется последовательность операторов, связанных с данным оператором **case**, пока не встретится оператор **break** или не будет достигнут конец оператора **switch**. Если значение выражения не совпадает ни с одной из констант, выполняется оператор **default**. Этот раздел оператора **switch** является необязательным. Если он не предусмотрен, в отсутствие совпадений не будет выполнен ни один оператор.

В языке C оператор **switch** допускает до 257 операторов **case**! Стандарт языка C++ предусматривает до 16384 операторов **case**! На практике количество разделов **case** в операторе **switch** следует ограничивать, поскольку оно влияет на эффективность программы. Несмотря на то что оператор **case** является меткой, он используется только внутри оператора **switch**.

Оператор **break** относится к группе операторов перехода. Его можно использовать как в операторе **switch**, так и в циклах (см. раздел "Операторы циклов"). Когда поток управления достигает оператора **break**, программа выполняет переход к оператору, следующему за оператором **switch**.

Следует знать три важных свойства оператора **switch**.

- Оператор **switch** отличается от оператора **if** тем, что значение его выражения сравнивается исключительно с константами, в то время как в операторе **if** можно выполнять какие угодно сравнения или вычислять любые логические выражения.
- Две константы в разных разделах **case** не могут иметь одинаковых значений, за исключением случая, когда один оператор **switch** вложен в другой.
- Если в операторе **switch** используются символьные константы, они автоматически преобразовываются в целочисленные.

Оператор **switch** часто используется для обработки команд, введенных с клавиатуры, например, при выборе пунктов меню. В приведенном ниже примере функция **menu()** выводит на экран меню программы для проверки правописания и вызывает соответствующие процедуры.

```
void menu(void)
{
  char ch;

  printf("1. Проверка правописания\n");
  printf("2. Исправление ошибок\n");
  printf("3. Вывод ошибок\n");
  printf("Для пропуска нажмите любую клавишу\n");
  printf("      Выберите пункт меню: ");

  ch = getchar(); /* Считывает символ, введенный с клавиатуры */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :
      printf("Не выбран ни один пункт");
  }
}
```

С формальной точки зрения наличие оператора **break** внутри оператора **switch** не обязательно. Этот оператор прерывает выполнение последовательности операторов, связанных с соответствующей константой. Если его пропустить, будут выполнены все последующие операторы **case**, пока не встретится следующий оператор **break**, либо не будет достигнут конец оператора **switch**. Например, приведенная ниже функция использует этот эффект для обработки информации, поступающей на вход драйвера.

```
/* Обработка значения */
void inp_handler(int i)
{
```

```
  int flag;

  flag = -1;

  switch(i) {
    case 1:  /* Эти операторы case имеют общую */
    case 2:  /* последовательность операторов. */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

Этот пример иллюстрирует два свойства оператора **switch**. Во-первых, оператор **case** может не иметь связанной с ним последовательности операторов. В этом случае поток управления просто переходит к следующему оператору **case**, как бы "проваливаясь" вниз. В нашем примере три первых оператора **case** связаны с одной и той же последовательностью операторов, а именно:

```
flag = 0;
break;
```

Во-вторых, если оператор **break** отсутствует, выполняется последовательность операторов, связанная со следующим оператором **case**. Если значение **i** равно 4, переменной **flag** присваивается число 1, и, поскольку оператора **break** в конце данного раздела **case** нет, выполнение оператора **switch** продолжается, и вызывается функция **error(flag)**. Если значение **i** равно 5, функция **error** будет вызвана с параметром **flag**, равным −1, а не 1.

То, что в отсутствие оператора **break** операторы **case** выполняются один за другим, позволяет избежать ненужного дублирования операторов и повысить эффективность программы.

Вложенные операторы switch

Операторы **switch** могут быть вложены друг в друга. Даже если константы разделов **case** внешнего и внутреннего операторов **switch** совпадают, проблемы не возникают. Например, приведенный ниже фрагмент программы является вполне приемлемым.

```
switch(x) {
  case 1:
    switch(y) {
      case 0: printf("Деление на нуль.\n");
              break;
      case 1: process(x,y);
    }
    break;
  case 2:
    .
    .
    .
```

Операторы цикла

В языке C/C++, как и во всех других современных языках программирования, операторы цикла предназначены для выполнения повторяющихся инструкций, пока действует определенное правило. Это условие может быть как задано заранее (в цикле **for**), так и меняться во время выполнения цикла (в операторах **while** и **do-while**).

Цикл for

В том или ином виде цикл **for** есть во всех процедурных языках программирования. Однако в языке C/C++ он обеспечивает особенно высокую гибкость и эффективность.

Общий вид оператора **for** таков.

for (*инициализация*; *условие*; *приращение*)

Цикл **for** имеет много вариантов. Однако наиболее общая форма этого оператора работает следующим образом. Сначала выполняется *инициализация* (initialization) — оператор присваивания, который задает начальное значение счетчика цикла. Затем проверяется *условие* (condition), представляющее собой условное выражение. Цикл выполняется до тех пор, пока значение этого выражения остается истинным. *Приращение* (increment) изменяет значение счетчика цикла при очередном его выполнении. Эти разделы оператора отделяются друг от друга точкой с запятой. Как только условие цикла станет ложным, программа прекратит его выполнение и перейдет к ледующему оператору.

В следующем примере цикл **for** выводит на экран числа от 1 до 100.

```
#include <stdio.h>

int main(void)
{
  int x;

  for(x=1; x <= 100; x++) printf("%d ", x);

  return 0;
}
```

Сначала переменной **x** присваивается число 1, а затем она сравнивается с числом 100. Поскольку ее значение меньше 100, вызывается функция **printf()**. Затем переменная **x** увеличивается на единицу, и условие цикла проверяется вновь. Как только ее значение превысит число 100, выполнение цикла прекратится. В данном случае переменная **x** является счетчиком цикла, который изменяется и проверяется на каждой итерации.

Рассмотрим пример цикла **for**, тело которого состоит из нескольких операторов.

```
for(x=100; x != 65; x -= 5) {
  z = x*x;
  printf("Квадрат числа %d равен %f", x, z);
}
```

Возведение числа **x** в квадрат и вызов функции **printf()** выполняются до тех пор, пока значение переменной **x** не станет равным 65. Обратите внимание на то, что

в этом цикле счетчик уменьшается: сначала ему присваивается число 100, а затем на каждой итерации из него вычитается число 5.

В цикле **for** проверка условия выполняется перед каждой итерацией. Иными словами, если условие цикла с самого начала является ложным, его тело не будет выполнено ни разу. Рассмотрим пример.

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y);  /* Это единственный вызов функции printf(),
                     который выполняется в данном фрагменте */
```

Этот цикл никогда не будет выполнен, поскольку значения переменных **x** и **y** при входе в цикл равны. Следовательно, условие цикла является ложным, и ни тело цикла, ни приращение счетчика выполняться не будут. Таким образом, значение переменной y останется равным 10, и именно оно будет выведено на экран.

Варианты цикла for

В предыдущем разделе описан наиболее общий вид оператора **for**. Однако он имеет несколько вариантов, повышающих его гибкость и эффективность.

Наиболее распространенным является вариант, в котором используется оператор последовательного выполнения ("запятая"), что позволяет применять несколько счетчиков цикла одновременно. (Напомним, что оператор последовательного выполнения связывает между собой несколько операторов, вынуждая их выполняться друг за другом. Подробности изложены в главе 2.) Например, переменные **x** и **y** являются счетчиками приведенного ниже цикла. Их инициализация выполняется в одном и том же разделе цикла.

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y - '0'; /* Вычесть из переменной y ASCII-код нуля */
    .


}
```

Как видим, два оператора инициализации разделены запятой. При каждой итерации значение переменной **x** увеличивается на единицу, а переменная **y** вводится с клавиатуры. Несмотря на это, переменная **y** должна иметь какое-то начальное значение, иначе перед первой итерацией цикла условие может оказаться ложным.

Функция **converge()**, приведенная ниже, демонстрирует одновременное применение нескольких счетчиков цикла. Она копирует одну строку в другую, перемещаясь от концов к середине.

```
/* Демонстрация одновременного использования
   нескольких счетчиков цикла */
#include <stdio.h>
#include <string.h>

void converge(char *targ, char *src);

int main(void)
{
  char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
                    "Проверка функции converge()."
  converge(target, "Проверка функции converge().");
```

```
  printf("Результат: %s\n", target);

  return 0;
}

/* Эта функция копирует одну строку в другую,
   перемещаясь от концов к середине. */
void converge(char *targ, char *src)
{
  int i, j;

  printf("%s\n", targ);
  for(i=0, j=strlen(src); i<=j; i++, j--) {
    targ[i] = src[i];
    targ[j] = src[j];
    printf("%s\n", targ);
  }
}
```

Программа выводит на экран следующие строки.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
ПXXXXXXXXXXXXXXXXXXXXXXXXXXX
ПрXXXXXXXXXXXXXXXXXXXXXXXXXX.
ПроXXXXXXXXXXXXXXXXXXXXXXXXX).
ПровXXXXXXXXXXXXXXXXXXXXXXX().
ПровеXXXXXXXXXXXXXXXXXXXXXe().
ПроверXXXXXXXXXXXXXXXXXXge().
ПроверкXXXXXXXXXXXXXXXXrge().
ПроверкаXXXXXXXXXXXXXXerge().
Проверка XXXXXXXXXXXverge().
Проверка фXXXXXXXXXnverge().
Проверка фуXXXXXXXonverge().
Проверка фунXXXXXconverge().
Проверка функXXX converge().
Проверка функцXи converge().
Проверка функции converge().
Результат: Проверка функции converge()
```

В функции **converge()** для индексации строки с обоих концов используются два счетчика цикла **for** — переменные **i** и **j**. При выполнении цикла счетчик **i** увеличивается, а счетчик **j** уменьшается. Выполнение цикла прекращается, когда значение счетчика **i** становится больше, чем значение счетчика **j**. В этот момент все символы строки уже скопированы.

Условное выражение не обязательно связано с проверкой счетчика цикла. В качестве условия цикла может использоваться любой допустимый оператор сравнения или логический оператор. Это позволяет задавать несколько условий цикла одновременно.

Рассмотрим функцию, которая проверяет пароль пользователя. Для ввода пароля пользователь может сделать три попытки. Выполнение цикла прекращается, если пользователь исчерпал все попытки или ввел правильный пароль.

```
void sign_on(void)
{
  char str[20];
  int x;

  for(x=0; x<3 && strcmp(str, "пароль"); ++x) {
```

```
    printf("Введите пароль:");
    gets(str);
  }

  if(x==3) return;
  /* Иначе пользователь получает доступ к системе ..  */
}
```

В этой функции используется стандартная функция **strcmp()**, сравнивающая две строки и возвращающая нуль, если они совпадают.

Напомним, что каждый из трех разделов цикла **for** может состоять из любых допустимых выражений. Эти выражения могут быть никак не связаны с предназначением разделов. Учитывая вышесказанное, рассмотрим следующий пример.

```
#include <stdio.h>

int sqrnum(int num);
int readnum(void);
int prompt(void);

int main(void)
{
  int t;

  for(prompt(); t=readnum(); prompt())
    sqrnum(t);

  return 0;
}

int prompt(void)
{
  printf("Введите число: ");
  return 0;
}

int readnum(void)
{
  int t;

  scanf("%d", &t);
  return t;
}

int sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```

Обратите внимание на цикл **for** в функции **main()**. Каждый из его разделов содержит вызов функции, в которой пользователю предлагается ввести с клавиатуры некое число. Если введено число 0, выполнение цикла прекращается, поскольку условное выражение становится ложным. В противном случае число возводится в квадрат. Таким образом, данный цикл **for** использует разделы инициализации и приращения крайне необычно, при этом и с синтаксической, и с семантической точки зрения цикл является абсолютно правильным.

Другая интересная особенность цикла **for** заключается в том, что его разделы можно пропускать. Каждый из его разделов является необязательным. Например, цикл, приведенный ниже, выполняется до тех пор, пока пользователь не введет число **123**:

```
for(x=0; x!=123; ) scanf("%d", &x);
```

Обратите внимание на то, что раздел приращения счетчика в данном цикле **for** отсутствует. Это значит, что при каждой итерации значение переменной **x** сравнивается с числом **123**, и никакие действия с ней больше не выполняются. Однако, если пользователь введет с клавиатуры число **123**, условие цикла станет ложным, и программа прекратит его выполнение.

Счетчик можно инициализировать вне цикла **for**. Этим способом пользуются, когда начальное значение счетчика является результатом сложных вычислений, как в следующем примере.

```
gets(s);  /* Считать строку в переменную s */
if(*s) x = strlen(s); /* Вычислить длину строки */
else x = 10;

for( ; x<10; ) {
  printf("%d", x);
  ++x;
}
```

Здесь раздел инициализации оставлен пустым, а переменная **x** инициализируется до входа в цикл.

Бесконечный цикл

Хотя в качестве бесконечного можно использовать любой цикл, традиционно для этой цели применяется оператор **for**. Поскольку все разделы оператора **for** являются необязательными, его легко сделать бесконечным, не задав никакого условного выражения.

```
for( ; ; ) printf("Этот цикл выполняется бесконечно.\n");
```

Если условное выражение не указано, оно считается истинным. Разумеется, в этом случае можно по-прежнему выполнять инициализацию и приращение счетчика, однако программисты на языке C++ в качестве бесконечного цикла чаще всего используют конструкцию **for(;;)**.

На самом деле конструкция **for(;;)** не гарантирует бесконечное выполнение цикла, поскольку его тело может содержать оператор **break**, приводящий к немедленному выходу. (Мы детально изучим этот оператор немного позднее.) В этом случае программа передаст управление следующему оператору, находящемуся за пределами тела цикла **for**, как показано ниже.

```
ch = '\0';

for( ; ; ) {
  ch = getchar(); /* Ввести символ */
  if(ch=='A') break; /* Выход из цикла */
}

printf("Вы ввели букву A");
```

Этот цикл выполняется до тех пор, пока пользователь не введет с клавиатуры букву A.

Пустой цикл for

Оператор может быть пустым. Это значит, что тело цикла **for** (как и любого другого цикла) может не содержать ни одного оператора. Этот факт можно использовать для повышения эффективности некоторых алгоритмов и задержки выполнения программы.

Удаление пробелов из входного потока — одна из наиболее распространенных задач. Например, система управления базой данных может допускать запрос "показать все счета, остаток на которых меньше 400". База данных распознает каждое слово отдельно, не учитывая пробелы. Следовательно, она распознает слово "показать", но не поймет слова " показать". Таким образом, пробелы в строке запроса необходимо игнорировать. Эту задачу решает цикл **for**, который пропускает все пробелы, стоящие перед словами в строке **str**.

```
for( ; *str == ' '; str++) ;
```

Как видим, этот цикл не имеет тела — оно ему не нужно.

Циклы часто используются для задержки выполнения программы. Ниже показано, как этого можно достичь, используя оператор **for**.

```
for(t=0; t<SOME_VALUE; t++) ;
```

Цикл while

Второй по значимости цикл в языке C/C++ — оператор **while**. Он имеет следующий вид.

while (*условие*) *оператор*;

Здесь *оператор* может быть пустым, отдельным оператором или блоком операторов. *Условие* может задаваться любым выражением. Условие цикла считается истинным, если значение этого выражения не равно нулю. Как только условие цикла становится ложным, программа передает управление оператору, стоящему сразу после оператора **while**.

Рассмотрим пример функции, обрабатывающей ввод с клавиатуры, которая терпеливо ждет, пока пользователь не введет букву A.

```
char wait_for_char(void)
{
  char ch;

  ch = '\0';  /* начальное значение переменной ch */
  while(ch != 'A') ch = getchar();
  return ch;
}
```

Сначала переменной **ch** присваивается нуль. Поскольку она является локальной, ее значение вне функции **wait_for_char()** не определено. Затем цикл **while** проверяет, не равно ли значение переменной **ch** символу A. Поскольку начальное значение переменной **ch** равно нулю, условие цикла является истинным, и его выполнение продолжается. Условие цикла проверяется каждый раз, когда пользователь вводит символ с клавиатуры. Как только он введет букву A, условие цикла станет ложным, и программа прекратит его выполнение.

Как и цикл **for**, цикл **while** проверяет свое условие перед началом выполнения тела. Следовательно, если условие цикла с самого начала является ложным, его тело не будет выполнено ни разу. Благодаря этому свойству нет необходимости отдельно проверять условие цикла перед входом в него. Эту особенность хорошо иллюстрирует функция **pad()**, добавляющая пробелы в конец строки, пока ее длина не станет равной заданной. Если длина строки уже равна требуемой величине, пробелы не добавляются.

```
#include <stdio.h>
#include <string.h>

void pad(char *s, int length);

int main(void)
{
  char str[80];

  strcpy(str, "Проверка");
  pad(str, 40);
  printf("%d", strlen(str));

  return 0;
}

/* Добавляет пробелы в конец строки. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s); /* Вычисляем длину строки */

  while(l<length) {
    s[l] = ' '; /* Вставляем пробел */
    l++;
  }
  s[l]= '\0'; /* Строка должна завершаться нулевым байтом */
}
```

Функция **pad()** имеет два аргумента — указатель **s** на удлиняемую строку и целочисленную переменную **length**, задающую требуемую длину строки. Если исходная длина строки равна числу **length** или превышает его, то тело цикла **while** не выполняется. Если же длина строки, на которую ссылается указатель **s**, меньше требуемой, функция **pad()** добавляет нужное количество пробелов. Длина строки вычисляется с помощью стандартной функции **strlen()**.

Если выход из цикла **while** зависит от нескольких условий, обычно в качестве условия цикла используют отдельную переменную, значение которой изменяется несколькими операторами в разных точках цикла. Рассмотрим пример.

```
void func1(void)
{
  int working;

  working = 1; /* т.е., истина */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

Каждая функция, вызванная в этом фрагменте программы, может вернуть ложное значение и привести к выходу из цикла.

Тело цикла **while** может быть пустым. Например, приведенный ниже пустой цикл выполняется до тех пор, пока пользователь не введет букву A.

```
while((ch=getchar()) != 'A') ;
```

Оператор присваивания внутри условного выражения цикла **while** выглядит непривычно, но все встанет на свои места, если вспомнить, что его значением является значение операнда, расположенного в правой части.

Цикл do-while

В отличие от операторов **for** и **while**, которые проверяют условие в начале цикла, оператор **do-while** делает это в конце. Иными словами, цикл **do-while** выполняется по крайней мере один раз. Общий вид оператора **do-while** таков.

do {
оператор;
} while(*условие***);**

Если тело цикла состоит лишь из одного оператора, фигурные скобки не обязательны, хотя они делают этот цикл понятнее (программисту, но не компилятору). Цикл **do-while** повторяется до тех пор, пока *условие* не станет ложным.

Рассмотрим пример, в котором цикл **do-while** считывает числа с клавиатуры, пока они не превысят 100.

```
do {
  scanf("%d", &num);
} while(num > 100);
```

Чаще всего оператор **do-while** применяется для выбора пунктов меню. Когда пользователь вводит допустимое значение, оно возвращается соответствующей функцией. Недопустимые значения игнорируются. Рассмотрим улучшенную версию программы для проверки правописания.

```
void menu(void)
{
  char ch;

  printf("1. Проверка правописания/n");
  printf("2. Исправление ошибок\n");
  printf("3. Вывод ошибок \n");
  printf("      Выберите пункт меню: ");

  do {
    ch = getchar(); /* Считываем символ с клавиатуры */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

Здесь оператор **do-while** очень удобен, поскольку функция должна вывести меню по крайней мере один раз. Если пользователь ввел допустимое значение, программа продолжит выполнять цикл.

Объявление переменных в условных операторах и циклах

В языке C++ (но не в стандарте C89) можно объявлять переменные внутри условных выражений, входящих в операторы **if** или **switch**, внутри условия цикла **while**, а также в разделе инициализации цикла **for**. Область видимости переменной, объявленной в одном из этих трех мест, ограничена блоком, который относится к данному оператору. Например, переменная, объявленная внутри цикла **for**, является локальной по отношению к нему.

Рассмотрим пример, в котором переменная объявлена в разделе инициализации цикла **for**.

```
/* Переменная i является локальной по отношению к циклу for,
   а переменная j существует как внутри, так и вне цикла. */
int j;
for(int i = 0; i<10; i++)
  j = i * i;

/* i = 10; // *** Ошибка *** -- переменная i здесь невидима! */
```

В этом примере переменная **i** объявлена внутри раздела инициализации цикла **for** и используется в качестве счетчика. Вне цикла **for** эта переменная не существует!

Поскольку счетчик нужен лишь внутри цикла, его объявление в разделе инициализации оператора **for** стало общепринятой практикой. Однако учтите, что стандартом C89 это не допускается. (В стандарте C99 это ограничение снято.)

Совет

*Утверждение, что переменная, объявленная в разделе инициализации цикла **for**, является локальной, в некоторых компиляторах не выполняется. В старых версиях такие переменные были доступны и вне оператора **for**. Однако стандарт языка C++ ограничил область видимости таких переменных телом оператора **for**.*

Если ваш компилятор полностью поддерживает стандарт языка C++, то переменные можно объявлять не только внутри циклов, но и внутри обычных условных выражений. Рассмотрим пример.

```
if(int x = 20) {
  x = x - y;
  if(x>10) y = 0;
}
```

Здесь объявляется переменная **x**, которой присваивается число 20. Поскольку это значение считается истинным, выполняется блок, связанный с оператором **if**. Область видимости переменной, объявленной внутри условного выражения, ограничивается этим блоком. Таким образом, вне оператора **if** переменной **x** не существует. Честно говоря, не все программисты считают приемлемым такой способ объявления переменных, поэтому в нашей книге мы не будем его применять.

Операторы перехода

В языке C/C++ предусмотрены четыре оператора безусловного перехода: **return**, **goto**, **break**, **continue**. Операторы **return** и **goto** можно применять в любом месте программы, в то время как операторы **break** и **continue** связаны с операторами циклов. Кроме того, оператор **break** можно применять внутри оператора **switch**.

Оператор return

Этот оператор используется для возврата управления из функции. Он относится к операторам безусловного перехода (jump operators), поскольку выполняет возврат в точку вызова функции. С ним может быть связано определенное значение, хотя это и не обязательно. Если оператор **return** связан с определенным значением, оно становится результатом функции. В стандарте C89 все функции, возвращающие значения, формально не обязаны были содержать оператор **return**. Если программист забывал указать его, функция возвращала какое-то случайное значение, так называемый "мусор". Однако в языке C++ (и в стандарте C99) все функции, возвращающие значения, *обязаны* содержать оператор **return**. Иными словами, если функция в языке C++ возвращает некое значение, любой оператор **return**, появляющийся в ее теле, должен быть связан с каким-то значением. (Даже в стандарте C89 требовалось, чтобы функция, возвращающая значение, действительно возвращала его!)

Оператор **return** имеет следующий вид.

return *выражение*;

Выражение указывается лишь тогда, когда в соответствии со своим объявлением функция возвращает какое-то значение. В этом случае результатом функции является значение данного *выражения*.

Внутри функции можно использовать сколько угодно операторов **return**. Однако функция прекратит свои вычисления, как только достигнет первого оператора **return**. Закрывающая фигурная скобка, ограничивающая тело функции, также приводит к прекращению ее выполнения. Она интерпретируется как оператор **return**, не связанный ни с каким значением. Если программист не укажет оператор **return** в функции, возвращающей некое значение, то ее результат останется неопределенным.

Функция, определенная со спецификатором **void**, может не содержать ни одного оператора **return**, связанного с каким-либо значением. Поскольку такие функции по определению не имеют возвращаемых значений, бессмысленно связывать с ними оператор **return**.

Более подробно оператор **return** рассматривается в главе 3.

Оператор goto

Поскольку в языке C/C++ существует богатый выбор управляющих структур на основе операторов **break** и **continue**, в операторе **goto** нет особой необходимости. Считается, что использование операторов **goto** снижает читабельность программ. Тем не менее, несмотря на то что оператор **goto** на протяжении многих лет подвергается остракизму, он по-прежнему иногда применяется. Трудно представить себе ситуацию, в которой нельзя было бы обойтись без оператора **goto**. И все же в умелых руках и при осторожном использовании оператор **goto** может принести пользу, например, при выходе из глубоко вложенных циклов. За рамками данного раздела в нашей книге вы больше нигде не встретите этот оператор.

Для оператора **goto** необходима *метка* (label), представляющая собой идентификатор с двоеточием. Метка должна находиться в той же функции, что и оператор **goto**, — перепрыгивать из функции в функцию нельзя. Оператор **goto** имеет следующий вид.

```
goto метка;
.
.
.
метка;
```

Метка может находиться как до, так и после оператора **goto**. Например, используя оператор **goto** и метку, можно организовать цикл от 1 до 100.

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

Оператор break

Оператор **break** применяется в двух ситуациях. Во-первых, он используется для прекращения выполнения раздела **case** внутри оператора **switch**. Во-вторых, с помощью оператора **break** можно немедленно выйти из цикла независимо от истинности или ложности его условия.

Если в цикле встречается оператор **break**, итерации прекращаются и выполнение программы возобновляется с оператора, следующего за оператором цикла. Рассмотрим пример.

```
#include <stdio.h>

int main(void)
{
  int t;

  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }

  return 0;
}
```

Эта программа выводит на экран числа от 0 до 10, а затем цикл прекращается, поскольку выполняется оператор **break**. Условие **t < 100** при этом игнорируется.

Оператор **break** часто применяется в циклах, выполнение которых следует немедленно прекратить при наступлении определенного события. В приведенном ниже примере нажатие клавиши прекращает выполнение функции **look_up()**.

```
void look_up(char *name)
{
  do {
    /* поиск имен ... */
    if(kbhit()) break;
  } while(!found);
  /* Точка выхода */
}
```

Если клавиша не нажата, функция **kbhit()** возвращает значение 0, в противном случае она возвращает ненулевое значение. Поскольку операционные системы значительно отличаются друг от друга, функция **kbhit()** не определяется стандартами языков C и C++, однако в большинстве компиляторов она предусмотрена.

Если циклы вложены друг в друга, оператор **break** выполняет выход из внутреннего цикла во внешний. Например, приведенная ниже программа 100 раз выводит на экран числа от 1 до 10, причем каждый раз, когда счетчик достигает значения 10, оператор **break** передает управление внешнему циклу **for**.

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

Если оператор **break** содержится внутри оператора **switch**, который вложен в некий цикл, то выход будет осуществлен только из оператора **switch**, а управление останется во внешнем цикле.

Функция exit

Хотя функция **exit()** не относится к управляющим операторам, настало время ее изучить. Вызов стандартной библиотечной функции **exit()** приводит к прекращению работы программы и передаче управления операционной системе. Ее эффект можно сравнить с катапультированием из программы.

Функция **exit()** выглядит следующим образом.

void exit(int *код_возврата*);

Значение переменной *код_возврата* передается вызывающему процессу, в роли которого чаще всего выступает операционная система. Нулевое значение кода возврата соответствует нормальному завершению работы. Другие значения аргумента указывают на вид ошибки. В качестве кода возврата можно применять макросы **EXIT_SUCCESS** и **EXIT_FAILURE**. Для вызова функции **exit()** необходим заголовочный файл **stdlib.h**. В программах на языке C++ можно также использовать заголовочный файл **<cstdlib>**.

Функция **exit()** часто используется, когда обязательное условие, гарантирующее правильную работу программы, не выполняется. Рассмотрим, например, виртуальную компьютерную игру, для которой требуется специальный графический адаптер. Функция **main()** в этой программе может выглядеть следующим образом.

```
#include <stdlib.h>

int main(void)
{
   if(!virtual_graphics()) exit(1);
   play();
   /* ... */
}
/* ...  */
```

Здесь функция **virtual_graphics()** определяется пользователем и возвращает истинное значение, когда в компьютере есть необходимый графический адаптер. Если же его нет, она возвращает ложное значение, и функция **exit()** прекращает работу программы

В программе для проверки правописания функция **menu()** может использовать функцию **exit()** для выхода из программы и возврата управления операционной системе.

```
void menu(void)
{
  char ch;

  printf("1. Проверка правописания\n");
  printf("2. Исправление ошибок\n");
  printf("3. Вывод ошибок\n");
  printf("4. Выход\n");
  printf("      Выберите пункт меню: ");

  do {
    ch = getchar(); /* Ввод символа с клавиатуры */
      switch(ch) {
        case '1':
          check_spelling();
          break;
        case '2':
          correct_errors();
          break;
        case '3':
          display_errors();
          break;
        case '4':
          exit(0); /* Возврат в операционную систему */
      }
    } while(ch!='1' && ch!='2' && ch!='3');
  }
```

Оператор continue

Оператор **continue** напоминает оператор **break**. Они различаются тем, что оператор **break** прекращает выполнение всего цикла, а оператор **continue** — лишь его текущей итерации, вызывая переход к следующей итерации и пропуская все оставшиеся операторы в теле цикла. В цикле **for** оператор **continue** вызывает проверку условия и приращение счетчика цикла. В циклах **while** и **do-while** оператор **continue** передает управление операторам, входящим в условие цикла. Приведенная ниже программа подсчитывает количество пробелов в строке, введенной пользователем.

```
/* Подсчет пробелов */
#include <stdio.h>

int main(void)
{
  char s[80], *str;
  int space;

  printf("Введите строку: ");
  gets(s);
  str = s;

  for(space=0; *str; str++) {
    if(*str != ' ') continue;
    space++;
  }
```

```
  printf("%d пробелов\n", space);

  return 0;
}
```

Проверяется каждый символ строки. Если он не является пробелом, оператор **continue** прерывает текущую итерацию цикла **for** и начинает новую, в противном случае значение счетчика **space** увеличивается на 1.

В следующем примере оператор **continue** выполняет выход из цикла **while**, передавая управление оператору, входящему в условие цикла.

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1); /* Перейти к следующей букве алфавита */
  }
}
```

Функция **code** кодирует сообщение, прибавляя единицу к коду каждого символа. Например, в сообщении на английском языке буква A заменяется буквой B и т.д. Функция прекращает свою работу, если пользователь ввел символ **$**. После этого вывод сообщений на экран прекращается, поскольку переменная **done** принимает истинное значение, и, соответственно, условие цикла становится ложным.

Операторы-выражения

Выражения были подробно рассмотрены в главе 2. Однако следует сделать несколько замечаний. Напомним, что любое допустимое выражение, завершающееся точкой с запятой, считается оператором. Например, операторами являются следующие выражения.

```
func();  /* Вызов функции */
a = b+c; /* Оператор присваивания */
b+f();   /* Нечто непонятное, но тоже допустимый оператор */
;        /* Пустой оператор */
```

Первый оператор выполняет вызов функции, второй — присваивание. Третий оператор выглядит весьма странно, но тем не менее успешно компилируется и выполняется (вызывается функция **f()** и ее результат суммируется со значением переменной **b**). Последний оператор является *пустым* (empty statement). (Иногда он также называется *фиктивным* (null statement).)

Блок

Блок — это группа связанных между собой операторов, рассматриваемых как единое целое. Операторы, образующие блок, логически связаны друг с другом. Блок иногда называется также *составным оператором*. Блок начинается открывающей фигурной скобкой **{** и завершается закрывающей фигурной скобкой **}**. Блок чаще всего используется

как составная часть другого оператора, например, условного оператора **if**. Однако блок может являться и самостоятельной единицей программы. Например, его можно использовать так, как показано в следующем примере (хотя обычно так не делают).

```
#include <stdio.h>

int main(void)
{
  int i;

  {  /* Блок */
     i = 120;
     printf("%d", i);
  }

  return 0;
}
```

Полный справочник по C++

Глава 4

Массивы и строки

Массив (array) — это совокупность переменных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется с помощью индекса. Согласно правилам языка C/C++ все массивы состоят из смежных ячеек памяти. Младший адрес соответствует первому элементу массива, а старший — последнему. Массивы могут быть одномерными и многомерными. Наиболее распространенным массивом является строка, завершающаяся нулевым байтом. Она представляет собой обычный массив символов, последним элементом которого является нулевой байт.

Массивы и указатели тесно связаны между собой. Трудно описывать массивы, не упоминая указатели, и наоборот. В этой главе мы опишем массивы, а указатели рассмотрим в главе 5. Чтобы хорошо разобраться в этих важных конструкциях, необходимо изучить обе главы.

Одномерные массивы

Объявление одномерного массива выглядит следующим образом.

тип имя_переменной[*размер*]

Как и другие переменные, массив должен объявляться явно, чтобы компилятор мог выделить память для него. Здесь *тип* объявляет базовый тип массива, т.е. тип его элементов, а *размер* определяет, сколько элементов содержится в массиве. Вот как выглядит объявление массива с именем **balance**, имеющего тип **double** и состоящего из 100 элементов.

```
double balance[100];
```

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Для этого индекс элемента указывается в квадратных скобках после имени массива. Например, оператор, приведенный ниже, присваивает третьему элементу массива **balance** значение 12.23.

```
balance[3] = 12.23;
```

Индекс первого элемента любого массива в языке C/C++ равен нулю. Следовательно, оператор

```
char p[10];
```

объявляет массив символов, состоящий из 10 элементов — от **p[0]** до **p[9]**. Следующая программа заполняет целочисленный массив числами от 0 до 99.

```
#include <stdio.h>

int main(void)
{
  int x[100]; /* Объявление целочисленного массива,
                 состоящего из 100 элементов */
  int t;

  /* Заполнение массива числами от 0 до 99 */
  for(t=0; t<100; ++t) x[t] = t;

  /* Вывод на экран элементов массива x */
  for(t=0; t<100; ++t) printf("%d ", x[t]);

  return 0;
}
```

Объем памяти, необходимый для хранения массива, зависит от его типа и размера. Размер одномерного массива в байтах вычисляется по формуле:

количество_байтов = **sizeof**(*базовый_тип*) * *количество_элементов*

В языке C/C++ не предусмотрена проверка выхода индекса массива за пределы допустимого диапазона. Иными словами, во время выполнения программы можно по ошибке выйти за пределы памяти, отведенной для массива, и записать данные в соседние ячейки, в которых могут храниться другие переменные и даже программный код. Ответственность за предотвращение подобных ошибок лежит на программисте. Например, фрагмент программы, приведенный ниже, будет скомпилирован без ошибок, однако во время выполнения программы индекс массива выйдет за пределы допустимого диапазона.

```
int count[10], i;

/* Выход индекса массива за пределы допустимого диапазона */
for(i=0; i<100; i++) count[i] = i;
```

По существу, одномерный массив представляет собой список переменных, имеющих одинаковый тип и хранящихся в смежных ячейках памяти в порядке возрастания их индексов. На рис. 4.1 показано, как хранится в памяти массив a, начинающийся с адреса 1000 и объявленный с помощью оператора

```
char a[7];
```

Элемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Адрес	1000	1001	1002	1003	1004	1005	1006

Рис. 4.1. Массив из семи символов, начинающийся с адреса 1000

Создание указателя на массив

Имя массива является указателем на первый его элемент. Допустим, массив объявлен с помощью оператора

```
int sample[10];
```

Указателем на его первый элемент при этом является имя **sample**. Таким образом, в следующем фрагменте указателю присваивается адрес первого элемента массива **sample**.

```
int *p;
int sample[10];

p = sample;
```

Адрес первого элемента массива можно также вычислить с помощью оператора **&**. Например, выражения **sample** и **&sample[0]** эквивалентны. Однако в профессионально написанных программах на языке C/C++ вы никогда не встретите выражение **&sample[0]**.

Передача одномерного массива в функцию

В языке C/C++ весь массив нельзя передать в качестве аргумента функции. Вместо этого можно передать указатель на массив, т.е. имя массива без индексов. Приведенный ниже фрагмент программы передает в функцию **func1()** индекс **i**.

```
int main(void)
{
  int i[10];

  func1(i);
  .
  .
  .
}
```

Если аргументом функции является одномерный массив, ее формальный параметр можно объявить тремя способами: как указатель, как массив фиксированного размера и как массив неопределенного размера. Например, чтобы предоставить функции **func1()** доступ к массиву **i**, можно использовать три варианта. Во-первых, в качестве ее аргумента можно объявить указатель.

```
void func1(int *x) /* Указатель */
{
  .
  .
  .
}
```

Во-вторых, можно передать указатель на массив фиксированного размера.

```
void func1(int x[10]) /* Массив фиксированного размера */
{
  .
  .
  .
}
```

И, наконец, можно использовать массив неопределенного размера.

```
void func1(int x[]) /* Массив неопределенного размера */
{
  .
  .
  .
}
```

Эти три объявления эквивалентны друг другу, поскольку их смысл совершенно одинаков: в функцию передается указатель на целочисленную переменную. В первом случае действительно используется указатель. Во втором применяется стандартное объявление массива. В последнем варианте объявляется, что в функцию будет передан целочисленный массив неопределенной длины. Размер массива, передаваемого в функцию, не имеет никакого значения, поскольку проверка выхода индекса за пределы допустимого диапазона в языке C/C++ не предусмотрена. При указании размера массива можно написать любое число — это ничего не изменит, поскольку в функцию будет передан не массив, состоящий из 32 элементов, а лишь указатель на его первый элемент.

```
void func1(int x[32]) /* Массив фиксированного размера */
{
  .
  .
  .
}
```

Строки, завершающиеся нулевым байтом

Чаще всего одномерный массив используется для представления строк символов. В языке C++ есть два вида строк. Во-первых, в нем предусмотрены *строки, завершающиеся нулевым байтом* (null-terminated string), представляющие собой массивы символов, последним элементом которых является нулевой байт. Это единственный вид строки, предусмотренный в языке C. Он по-прежнему является наиболее широко распространенным видом строк. Иногда строки, завершающиеся нулевым байтом, называются *C-строками* (C-string). В языке C++ определен также класс **string**, который реализует объектно-ориентированный подход к обработке строк. Этот класс мы рассмотрим позднее, а пока сосредоточимся на строках, завершающихся нулевым байтом.[4]

Объявляя массив символов, следует зарезервировать одну ячейку для нулевого байта, т.е. указать размер, на единицу больше, чем длина наибольшей предполагаемой строки. Например, чтобы объявить массив **str**, предназначенный для хранения строки, состоящей из 10 символов, следует выполнить следующий оператор.

```
char str[11];
```

В этом объявлении предусмотрена ячейка, в которой будет записан нулевой байт.

Строка символов, заключенных в двойные кавычки, например, "Здравствуйте, я ваша тетя!", называется *строковой константой* (string constant). Компилятор автоматически добавляет к ней нулевой байт, поэтому программист может не беспокоиться об этом.

В языке C/C++ предусмотрен богатый выбор функций для работы со строками. Самыми распространенными среди них являются следующие.

Имя	Предназначение
strcpy(s1,s2)	Копирует строку **s2** в строку **s1**.
strcat(s1,s2)	Приписывает строку **s2** в конец строки **s1**.
strlen(s1)	Вычисляет длину строки **s2**.
strcmp(s1,s2)	Возвращает 0, если строки **s1** и **s2** совпадают, отрицательное значение, если **s1<s2**, и положительное значение, если **s1>s2**.
strchr(s1,ch)	Возвращает указатель на позицию первого вхождения символа **ch** в строку **s1**.
strstr(s1,s2)	Возвращает указатель на позицию первого вхождения строки **s2** в строку **s1**.

Эти функции объявлены в стандартном заголовочном файле **string.h**. (В программах на языке C++ используется также заголовочный файл **<cstring>**.) Применение этих функций иллюстрируется следующей программой.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);
```

[4] Не следует путать нуль (**0**) и нулевой байт (**\0**), или нулевой символ. Признаком конца строки является именно нулевой байт. Если не указать обратную косую черту, нуль будет считаться обычным символом. — *Прим. ред.*

```
  printf("Длина строк: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf("Строки одинаковы\n");

  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "Проверка.\n");
  printf(s1);
  if(strchr("Привет", 'e'))
     printf("В слове \"Привет\" есть буква \"e\"\n");
  if(strstr("Привет", "ив"))
     printf("В слове \"Привет\" есть сочетание букв \"ив\"");

  return 0;
}
```

Если запустить на выполнение эту программу и ввести строки "Привет" и "Привет", то на экран будут выведены следующие сообщения:

```
Длина строк: 6 6
Строки одинаковы
ПриветПривет
Проверка
В слове "Привет" есть буква "e"
В слове "Привет" есть сочетание букв "ив"
```

Обратите внимание на то, что функция **strcmp()** возвращает ложное значение, только если строки одинаковы. Если необходимо проверить, совпадают ли строки между собой, следует использовать логический оператор отрицания "**!**".[5]

Несмотря на то что в языке C++ для строк предусмотрен отдельный класс, строки, завершающиеся нулевым байтом, по-прежнему весьма популярны. Возможно, это происходит благодаря тому, что они очень эффективны и позволяют программистам полностью контролировать операции над строками. Однако во многих ситуациях очень удобно использовать класс **string**.

Двухмерные массивы

В языке C/C++ предусмотрены многомерные массивы. Простейшим из них является двухмерный. По существу, двухмерный массив — это массив одномерных массивов. Объявление двухмерного массива **d**, состоящего из 10 строк и 20 столбцов, выглядит следующим образом.

```
int d[10][20];
```

Объявляя двухмерные массивы, следует проявлять осторожность. В некоторых языках программирования размерности массивов отделяются запятыми. В отличие от них, в языке C/C++ размерности массива заключаются в квадратные скобки.

Обращение к элементу двухмерного массива выглядит так:

```
d[1][2]
```

[5] Учтите также, что отношения "больше" и "меньше" между строками понимаются в лексикографическом смысле. Например, значение **strcmp("A","Я")** равно −31 (т.е. строка **"A"** меньше строки **"Я"**), а значение **strcmp("Я","A")** равно 31. Обратите внимание на то, что значение **strcmp("АЯ", "ЯА")** снова равно −31, иными словами, функция **strcmp** возвращает разность между ASCII-кодами первых несовпадающих между собой символов. Собственно, именно так упорядочиваются слова в алфавитном порядке. — *Прим. ред.*

Следующая программа заполняет двухмерный массив числами от 1 до 19 и выводит их на экран построчно.

```
#include <stdio.h>

int main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

  /* Вывод на экран */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }

  return 0;
}
```

В данном примере элемент **num[0][0]** равен 1, **num[0][1]** равен 2, **num[0][2]** равен 3 и т.д. Значение элемента **num[2][3]** равно 12. Массив **num** можно изобразить следующим образом.

num[t][i]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	8	10	11	12

Двухмерный массив хранится в виде матрицы, в которой первый индекс задает номер строки, а второй — номер столбца. Таким образом, при обходе элементов в порядке их размещения в памяти правый индекс изменяется быстрее, чем левый. Графическая схема размещения двухмерного массива в памяти показана на рис. 4.2.

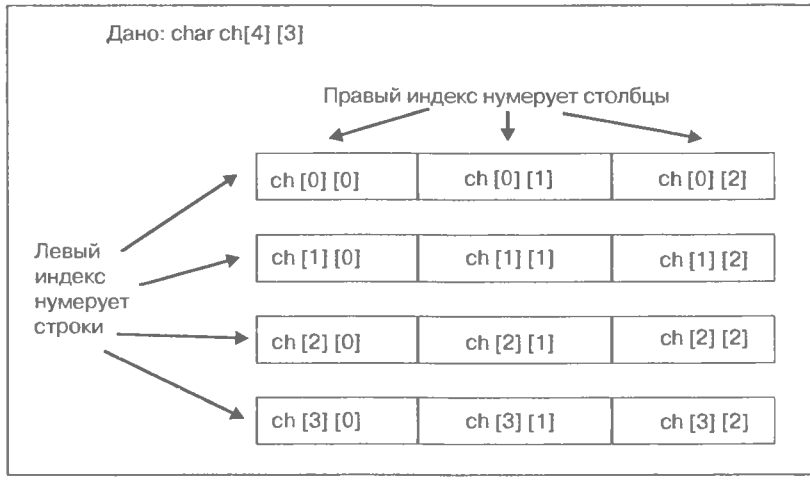


Рис. 4.2. Двухмерный массив

Объем памяти, занимаемый двухмерным массивом, выраженный в байтах, задается следующей формулой:

количество_байтов = *количество_строк***количество_столбцов****sizeof**(*базовый_тип*)

Считая, что целое число занимает 4 байт, можно вычислить, что для хранения массива, состоящего из 10 строк и 5 столбцов, необходимо 10*5*4=200 байт.

Если двухмерный массив используется в качестве аргумента функции, то в нее передается только указатель на его первый элемент. Однако при этом необходимо указать, по крайней мере, количество столбцов. (Можно, разумеется, задать и количество строк, но это не обязательно.) Количество столбцов необходимо компилятору для того, чтобы правильно вычислить адрес элемента массива внутри функции, а для этого должна быть известна длина строки. Например, функция, получающая в качестве аргумента двухмерный массив, состоящий из 10 строк и 10 столбцов, может выглядеть так.

```
void func1(int x[][10])
{



}
```

Компилятор должен знать количество столбцов, иначе он не сможет правильно вычислять выражения, подобные следующему:

```
x[2][4]
```

Если бы длина строки была неизвестна, компилятор не нашел бы начало третьей строки.

Рассмотрим короткую программу, в которой двухмерный массив используется для хранения оценок, полученных студентами. Предполагается, что учитель преподает в трех группах, в которых учатся не более 30 студентов. Обратите внимание на то, как происходит обращение в массиву **grade** в каждой функции.

```
/* Простая база данных, содержащая оценки студентов. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define CLASSES  3
#define GRADES  30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

int main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(В)вод оценок \n");
      printf("(П)ечать оценок\n");
      printf("(К)онец\n");
      gets(str);
```

```
      ch = toupper(*str);
    } while(ch!='B' && ch!='П' && ch!='K');

    switch(ch) {
      case 'B':
        enter_grades();
        break;
      case 'П':
        disp_grades(grade);
        break;
      case 'K':
        exit(0);
    }
  }

  return 0;
}

/* Ввод оценок. */
void enter_grades(void)
{
  int t, i;

  for(t=0; t<CLASSES; t++) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i] = get_grade(i);
  }
}

/* Считывание оценки. */
int get_grade(int num)
{
  char s[80];

  printf("Введите оценку студента # %d:\n", num+1);
  gets(s);
  return(atoi(s));
}

/* Вывод оценок. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Группа # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Студент #%d is %d\n", i+1, g[t][i]);
  }
}
```

Массивы строк

Массивы строк используются довольно часто. Например, сервер базы данных может сравнивать команды, введенные пользователем, с массивом допустимых команд. Для того чтобы создать массив строк, завершающихся нулевым байтом, необходим двухмерный массив символов. Максимальное значение левого индекса задает количество строк, а правого индекса — максимальную длину каждой строки. В приведенном

ниже фрагменте объявлен массив, состоящий из 30 строк, каждая из которых может содержать до 79 символов и нулевой байт.

```
char str_array[30][80];
```

Доступ к отдельной строке весьма прост: нужно лишь указать левый индекс. Например, следующий оператор применяет функцию **gets()** к третьей строке массива **str_array**.

```
gets(str_array[2]);
```

С функциональной точки зрения этот оператор эквивалентен такому вызову:

```
gets(&str_array[2][0]);
```

И все же первая из этих форм записи предпочтительнее.

Чтобы лучше понять, как устроен массив строк, рассмотрим короткую программу, в которой массив строк используется как основа для текстового редактора.

```
/* Очень простой текстовый редактор. */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
  register int t, i, j;

  printf("Для выхода введите пустую строку.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* Выход */
  }

  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++) putchar(text[i][j]);
    putchar('\n');
  }

  return 0;
}
```

Ввод текста в эту программу продолжается до тех пор, пока пользователь не введет пустую строку. Затем все строки выводятся на экран по очереди.

Многомерные массивы

В языке C/C++ массивы могут иметь больше двух размерностей. Максимально допустимое количество размерностей задается компилятором. Общий вид объявления многомерного массива таков.

тип имя[*Размер1*] [*Размер2*] [*Размер3*]... [*РазмерN*]

Массивы, имеющие больше трех размерностей, используются редко, поскольку они занимают слишком большой объем памяти. Например, четырехмерный массив симво-

лов размерностью 10×6×9×4 занимает 2160 байт. Если бы массив содержал двухбитовые целые числа, потребовалось бы 4320 байт. Если бы элементы массива имели тип **double** и занимали бы 8 байт каждый, то для массива потребовалось бы 17280 байт. Размер памяти, выделяемой для массива, экспоненциально возрастает с увеличением количества размерностей, например, если к предыдущему массиву добавить пятое измерение, в котором располагается 10 элементов, то для него понадобилось бы 172800 байт.

При обращении к многомерным массивам компилятор вычисляет каждый индекс. Следовательно, доступ к элементам многомерного массива происходит значительно медленнее, чем к элементам одномерного массива.

Передавая многомерный массив в функции, следует указывать все размеры, кроме первого. Например, если массив **m** объявлен оператором

```
int m[4][3][6][5];,
```

то функция **func1()**, получающая его в качестве аргумента, может выглядеть так:

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

Разумеется, при желании можно указать и первый размер, но это не обязательно.

Индексация указателей

В языке C/C++ указатели и массивы тесно связаны друг с другом. Как известно, имя массива без индекса — это указатель на его первый элемент. Рассмотрим, например, следующий массив.

```
char p[10];
```

Следующие два выражения абсолютно идентичны.

```
p
&p[0]
```

С другой стороны, выражение

```
p == &p[0]
```

является истинным, поскольку адрес первого элемента массива совпадает с адресом всего массива.

Как указывалось ранее, имя массива представляет собой указатель. И, наоборот, указатель можно индексировать, как массив. Рассмотрим следующий фрагмент программы.

```
int *p, i[10];
p = i;
p[5] = 100;    /* Присваивание с помощью индекса */
*(p+5) = 100; /* Присваивание с помощью адресной арифметики */
```

Оба оператора присваивания заносят число 100 в шестой элемент массива **i**. Первый из них индексирует указатель **p**, а второй использует адресную арифметику. В обоих случаях результат одинаков. (Указатели и адресная арифметика изучаются в главе 5.)

Этот способ можно применять и к многомерным массивам. Например, если переменная **a** — это указатель на целочисленный массив, состоящий из 10 строк и 10 столбцов, то следующие два оператора будут эквивалентными.

```
a
&a[0][0]
```

Более того, доступ к элементу, стоящему на пересечении первой строки и пятого столбца, можно получить двумя способами: либо индексируя массив — **a[0][4]**, либо используя указатель — ***((int *)a+4)**. Аналогично элемент, стоящий во второй строке и третьем столбце, является значением выражений **a[1][2]** и ***((int *)a+12)**. Для двухмерного массива справедлива следующая формула:

a[j][k] =* ((*базовый_тип* ***)a** + (**i** * *длина_строки*) + **k**)

Правила адресной арифметики требуют приведения типа указателя на массив к его базовому типу. Обращение к элементам массива с помощью указателей используется довольно широко, поскольку операции адресной арифметики выполняются быстрее, чем индексация.

Двухмерный массив можно представить с помощью указателя на массив одномерных массивов. Следовательно, используя отдельный указатель, можно обращаться к элементам, стоящим в строках двухмерного массива. Этот способ иллюстрируется следующей функцией. Она выводит на печать содержимое заданной строки глобального целочисленного массива **num**.

```
int num[10][10];
  .
  .
  .
void  pr_row(int j)
{
  int *p, t;

  p = (int *) &num[j][0]; /* Получить адрес первого элемента
                             в строке с индексом j */

  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

Эту функцию можно обобщить, включив в список аргументов индекс строки, ее длину и указатель на первый элемент массива.

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;

  p = p + (j * row_dimension);

  for(t=0; t<row_dimension; ++t)
    printf("%d ", *(p+t));
}
  .
  .
  .
void f(void)
{
  int num[10][10];

  pr_row(0, 10, (int *) num); /* Вывод первой строки */
}
```

Этот способ можно применять и к массивам более высокой размерности. Например, трехмерный массив можно свести к указателю на двухмерный массив, который, в свою очередь, можно представить с помощью указателя на одномерный массив. В общем случае *n*-мерный массив можно свести к указателю на (n-1)-мерный массив и т.д. Процесс завершается получением указателей на одномерный массив.

Инициализация массива

В языке C/C++ допускается инициализация массивов при их объявлении. Общий вид инициализации массива не отличается от инициализации обычных переменных.

тия_массива имя_массива[*размер1*]...[*размерN*] = {*список_значений*}

Список_значений представляет собой список констант, разделенных запятыми. Тип констант должен быть совместимым с *типом массива*. Первая константа присваивается первому элементу массива, вторая — второму и т.д. Обратите внимание на то, что после закрывающейся фигурной скобки **}** обязательно должна стоять точка с запятой.

Рассмотрим пример, в котором целочисленный массив, состоящий из 10 элементов, инициализируется числами от 1 до 10.

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Здесь элементу **i[0]** присваивается значение 1, а элементу **i[9]** — число 10.

Символьные массивы можно инициализировать строковыми константами:

char *имя_массива*[*размер*]="*строка*";

Например, следующий фрагмент инициализирует массив **str** строкой "Я люблю C++".

```
char str[12] = "Я люблю C++";
```

То же самое можно переписать иначе.

```
char str[12] = {'Я', ' ', 'л', 'ю', 'б', 'л', 'ю',' ', 'C',
                '+', '+', '\0'};
```

Поскольку строки завершаются нулевым байтом, необходимо следить, чтобы размер массива был достаточным. Вот почему размер массива **str** равен 12, хотя строка "Я люблю C++" состоит из 11 символов. Если массив инициализируется строковой константой, компилятор автоматически добавляет нулевой байт.

Многомерные массивы инициализируются аналогично. Рассмотрим пример инициализации массива **sqrs** числами от 1 до 10 и их квадратами.

```
int sqrs[10][2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100
};
```

Инициализируя многомерный массив, можно заключать элементы списка в фигурные скобки. Этот способ называется *субагрегатной группировкой* (subaggregate grouping). Например, предыдущую инициализацию можно переписать следующим образом.

```
int sqrs[10][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
  {8, 64},
  {9, 81},
  {10, 100}
};
```

Если при группировке данных указаны не все начальные значения, оставшиеся элементы группы автоматически заполняются нулями.

Инициализация безразмерного массива

Представьте себе инициализацию таблицы сообщений об ошибках, каждое из которых хранится в одномерном массиве.

```
char e1[12] = "Ошибка при чтении\n";
char e2[13] = "Ошибка при записи\n";
char e3[18] = "Невозможно открыть файл\n";
```

Чтобы задать правильный размер массивов, пришлось бы подсчитывать точное количество символов в каждом сообщении. К счастью, компилятор может сам определить размер массива. Если в операторе инициализации не указан размер массива, компилятор автоматически создаст массив, вмещающий все начальные значения. Такой массив называется *безразмерным* (unsized array). Используя этот подход, можно переписать предыдущий фрагмент программы иначе.

```
char e1[] = "Ошибка при чтении\n";
char e2[] = "Ошибка при записи\n";
char e3[] = "Невозможно открыть файл\n";
```

В этом случае оператор

```
printf("Длина строки \"%s\" равна %d\n",  e2,  sizeof e2);
```

выведет на экран сообщение

```
Длина строки "Ошибка при записи" равна 18
```

Инициализация безразмерных массивов не только облегчает программирование, но и позволяет изменять сообщения, не заботясь о размере массивов.

Этот способ применим и к многомерным массивам. Для этого требуется указать все размеры, кроме первого. (Остальные размеры нужны для того, чтобы компилятор правильно выполнял индексацию массива.) Это позволяет создавать таблицы переменной длины, поскольку компилятор автоматически размещает их в памяти. Например, объявление массива **sqrt** как безразмерного выглядит так:

```
int sqrs[][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
```

```
  {8, 64},
  {9, 81},
  {10, 100}
};
```

Преимущество такого объявления состоит в том, что размер таблицы можно изменять, не меняя размеры массива.

Игра в крестики-нолики

Приведенный ниже пример иллюстрирует способы манипулирования массивами в языке C/C++. В этом разделе описана простая программа для игры в крестики-нолики. В качестве игральной доски используется двухмерный массив.

Компьютер играет очень просто. Когда наступает его очередь ходить, он просматривает матрицу с помощью функции **get_computer_move()** в поисках свободной ячейки. Найдя такую ячейку, заполняет ее символом **O**. Если все ячейки заняты, фиксируется ничья, и программа прекращает свою работу. Функция **get_player_move()** предлагает игроку сделать очередной ход и занести символ **X** в указанную им ячейку. Координаты левого верхнего угла равны (1, 1), а правого нижнего — (3, 3).

Массив **matrix** инициализируется пробелами. Каждый ход, сделанный игроком или компьютером, заменяет пробел в соответствующей ячейке символами **O** или **X**. Это позволяет легко вывести матрицу на экран.

После каждого хода вызывается функция **check()**. Если победитель еще не определен, она возвращает пробел, если выиграл игрок — символ **X**, а если компьютер — символ **O**. Эта функция просматривает матрицу в поисках строк, столбцов и диагоналей, заполненных одинаковыми символами.

Текущее состояние игры отображается функцией **disp_matrix()**. Обратите внимание, насколько инициализация матрицы пробелами упрощает эту функцию.

Все функции получают доступ к массиву **matrix** по-разному. Изучите эти способы внимательно, чтобы лучше понять их механизм.

```
/* Игра в крестики-нолики. */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3];  /* Игральная доска */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

int main(void)
{
  char done;

  printf("Это - игра в крестики нолики.\n");
  printf("Вы будете играть с компьютером.\n");

  done = ' ';
  init_matrix();
  do{
    disp_matrix();
```

```
    get_player_move();
    done = check(); /* Есть ли победитель? */
    if(done!= ' ') break; /* Победитель определен!*/
    get_computer_move();
    done = check(); /* Есть ли победитель? */
  } while(done== ' ');
  if(done=='X') printf("Вы победили!\n");
  else printf("Я выиграл!!!!\n");
  disp_matrix(); /* Финальное положение */

  return 0;
}

/* Инициализация матрицы. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++) matrix[i][j] =  ' ';
}

/* Ход игрока. */
void get_player_move(void)
{
  int x, y;

  printf("Введите координаты X,Y: ");
  scanf("%d%*c%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!= ' '){
    printf("Неверный ход, попытайтесь еще.\n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}

/* Ход компьютера. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }

  if(i*j==9)  {
    printf("Ничья\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
}

/* Вывести матрицу на экран. */
```

```
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c | %c | %c ",matrix[t][0],
            matrix[t][1], matrix [t][2]);
    if(t!=2) printf("\n---|---|---\n");
  }
  printf("\n");
}

/* Проверить, есть ли победитель. */
char check(void)
{
  int i;

  for(i=0; i<3; i++)  /* Проверка строк */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matrix[i][2]) return matrix[i][0];

  for(i=0; i<3; i++)  /* Проверка столбцов */
    if(matrix[0][i]==matrix[1][i] &&
       matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* Проверка диагоналей */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matrix[2][2])
       return matrix[0][0];

  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matrix[2][0])
       return matrix[0][2];

  return ' ';
}
```

Полный справочник по

Глава 5

Указатели

Существуют три причины, по которым невозможно написать хорошую программу на языке C/C++ без использования указателей. Во-первых, указатели позволяют функциям изменять свои аргументы. Во-вторых, с помощью указателей осуществляется динамическое распределение памяти. И, в-третьих, указатели повышают эффективность многих процедур. Как мы увидим в дальнейшем, в языке C++ указатели обладают дополнительными преимуществами.

Указатели — одно из самых мощных и, в то же время, самых опасных средств языка C/C++. Например, неинициализированные указатели (или указатели, содержащие неверные адреса) могут уничтожить операционную систему компьютера. И, что еще хуже, неправильное использование указателей порождает ошибки, которые крайне трудно обнаружить.

Поскольку указатели настолько важны и опасны, мы рассмотрим их весьма подробно.

Что такое указатель

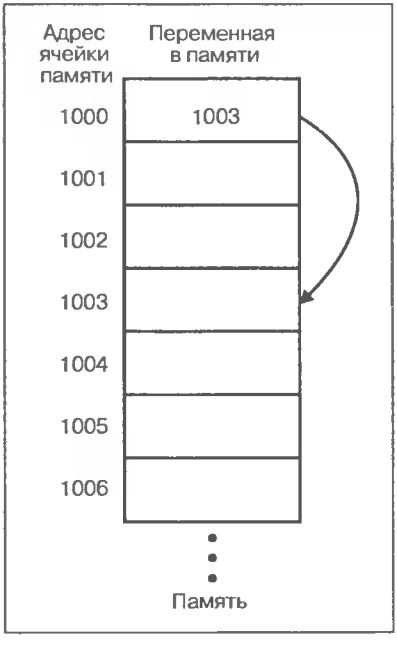


Рис. 5.1. Одна переменная ссылается на другую

Указатель (pointer) — это переменная, в которой хранится адрес другого объекта (как правило, другой переменной). Например, если одна переменная содержит адрес другой переменной, говорят, что первая переменная *ссылается* (point) на вторую. Эта ситуация показана на рис. 5.1.

Указатели

Переменная, хранящая адрес ячейки памяти, должна быть объявлена как указатель. Объявление указателя состоит из имени базового типа, символа * и имени переменной. Общая форма этого объявления такова.

тип_указателя ***имя_указателя***;

Здесь *тип_указателя* означает базовый тип указателя. Им может быть любой допустимый тип.

Базовый тип указателя определяется типом переменной, на которую он может ссылаться. С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей, поэтому очень важно правильно их объявить. (Адресная арифметика обсуждается ниже.)

Операторы для работы с указателями

Эти операторы обсуждались в главе 2. Рассмотрим их повнимательнее. Как известно, существуют два специальных оператора для работы с указателями: оператор разыменования указателя * и оператор получения адреса &. Оператор & является унарным и возвращает адрес своего операнда. (Не забывайте: унарные операторы имеют только один операнд.) Например, оператор присваивания

```
m = &count;
```

записывает в указатель **m** адрес переменной **count**. Этот адрес относится к ячейке памяти, которую занимает переменная **count**[6]. Адрес и значение переменной никак не связаны друг с другом. Оператор & означает "адрес". Следовательно, предыдущий оператор присваивания можно прочитать так: "присвоить указателю **m** адрес переменной **count**".

Чтобы глубже разобраться в механизме, лежащем в основе предыдущего оператора присваивания, предположим, что переменная **count** хранится в ячейке с номером 2000.

Оператор разыменования указателя * является антиподом оператора &. Этот унарный оператор возвращает значение, хранящееся по указанному адресу. Например, если указатель **m** содержит адрес переменной **count**, то оператор присваивания

```
q = *m;
```

поместит значение **count** в переменную **q**. Следовательно, переменная **q** станет равной 100, поскольку именно это число записано в ячейке 2000, адрес которой хранится в указателе **m**. Символ * можно интерпретировать как "значение, хранящееся по адресу". В данном случае предыдущий оператор означает: "присвоить переменной **q** значение, хранящееся по адресу **m**".

Приоритет операторов & и * выше, чем приоритет всех арифметических операторов, за исключением унарного минуса.

Необходимо иметь гарантии, что указатель всегда ссылается на переменную правильного типа. Например, если в программе объявлен указатель на целочисленную переменную, компилятор полагает, что адрес, который в нем содержится, относится к переменной типа **int**, независимо от того, так ли это на самом деле. Поскольку указателю можно присвоить любой адрес, следующая программа будет скомпилирована без ошибок, но желаемого результата не даст.

```
#include <stdio.h>

int main(void)
{
  double x = 100.1, y;
  int  *p;

  /* Следующий оператор заставит указатель p (имеющий
     целочисленный тип) ссылаться на переменную типа double. */
  p = &x;
```

[6] Если переменная занимает несколько ячеек памяти, ее адресом считается адрес первой ячейки. — *Прим. ред.*

```
  /* Следующий оператор работает не так, как задумано. */
  y = *p;


  printf("%f", y); /* Не выводит число 100.1 */
  return 0;
}
```

Эта программа никогда не присвоит переменной **y** значение переменной **x**. Поскольку указатель **p** является целочисленным, в переменную **y** будут скопированы лишь 4 байт (поскольку целые числа занимают 4 байт), а не 8.

На заметку *В языке C++ нельзя изменить тип указателя, не используя явное приведение типов. В языке C приведение типов используется в большинстве случаев.*

Выражения, содержащие указатели

Как правило, выражения, содержащие указатели, подчиняются общепринятым правилам. Однако у них есть свои особенности.

Присваивание указателей

Указатель можно присваивать другому указателю. Рассмотрим пример.

```
#include <stdio.h>

int main(void)
{
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf(" %p", p2); /* Выводит адрес переменной x,
                        а не ее значение! */

  return 0;
}
```

Теперь на переменную **x** ссылаются оба указателя **p1** и **p2**. Адрес переменной **x** выводится на экран с помощью форматного спецификатора **%p** и функции **printf()**.

Адресная арифметика

К указателям можно применять только две арифметические операции: сложение и вычитание. Предположим, что указатель **p1** ссылается на целочисленную переменную, размещенную по адресу 2000. Кроме того, будем считать, что целые числа занимают 2 байт в памяти компьютера. После вычисления выражения

```
p1++;
```

переменная **p1** будет равна не 2001, а 2002, поскольку при увеличении указателя **p1** на единицу он ссылается на следующее целое число. Это же относится и к оператору декрементации. Например, если указатель **p1** равен 2000, выражение

```
p1--;
```

присвоит указателю **p1** значение 1998.

Обобщая приведенный выше пример, сформулируем следующие правила адресной арифметики. При увеличении указатель ссылается на ячейку, в которой хранится следующий элемент базового типа. При уменьшении он ссылается на предыдущий элемент. Для указателей на символы сохраняются правила "обычной" арифметики, поскольку размер символов равен 1 байт. Все остальные указатели увеличиваются или уменьшаются на длину соответствующих переменных, на которые они ссылаются. Это гарантирует, что указатели всегда будут ссылаться на элемент базового типа. Описанная выше концепция проиллюстрирована на рис. 5.2.

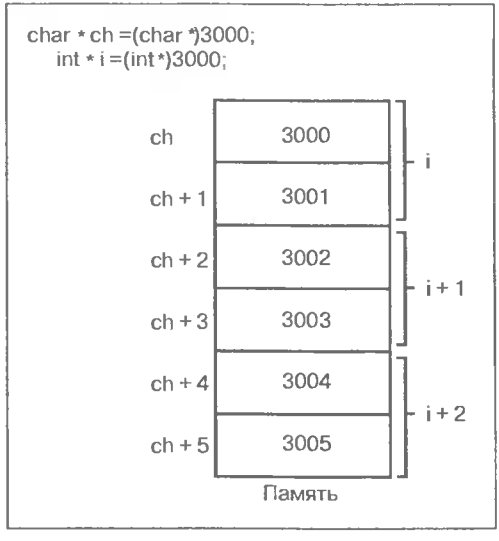


Рис. 5.2. Адресная арифметика тесно связана с базовым типом (предполагается, что длина целочисленной переменной равна двум байтам)

Действия над указателями не ограничиваются операторами инкрементации и декрементации. Например, к ним можно добавлять и целые числа. Выражение

```
p1 = p1 + 12;
```

смещает указатель **p1** на 12-й элемент базового типа, расположенный после текущего адреса.

Кроме того, указатели можно вычитать. Это позволяет определить количество объектов базового типа, расположенных между двумя указателями. Все другие арифметические операции запрещены. В частности, указатели нельзя складывать, умножать и делить. К ним нельзя применять побитовые операции, суммировать их со значениями переменных, имеющих тип **float**, **double** и т.п., а также вычитать такие значения из них.

Сравнение указателей

Указатели можно сравнивать между собой. Например, следующий оператор, в котором сравниваются указатели **p** и **q**, является совершенно правильным.

```
if(p<q) printf("Указатель p содержит меньший адрес,
                чем указатель q\n");
```

Как правило, указатели сравниваются между собой, когда они ссылаются на один и тот же объект, например массив. Рассмотрим в качестве примера пару функций для работы со стеками, которые записывают и извлекают целые числа. Стек — это список, в котором доступ к элементам осуществляется по принципу "первым вошел, последним вышел". Его часто сравнивают со стопкой тарелок на столе — нижняя тарелка будет снята последней. Стеки используются в компиляторах, интерпретаторах, программах обработки электронных таблиц и других системных утилитах. Чтобы создать стек, необходимы две функции: **push()** и **pop()**. Функция **push()** заносит числа в стек, а функция **pop()** извлекает их оттуда. В приведенной ниже программе они управляются функцией **main()**. При вводе числа с клавиатуры программа заталкивает его в стек. Если пользователь ввел число **0**, значение выталкивается из стека. Программа прекратит свою работу, когда пользователь введет число −1.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 50

void push(int i);
int pop(void);

int  *tos, *p1, stack[SIZE];

int main(void)
{
  int value;

  tos = stack; /* Указатель tos ссылается на вершину стека */
  p1 = stack; /* Инициализация указателя p1 */

  do {
    printf("Введите число: ");
    scanf("%d", &value);
    if(value!=0) push(value);
    else printf("Число на вершине стека равно %d\n", pop());
  } while(value!=-1);

  return 0;
}

void push(int i)
{
  p1++;
  if(p1==(tos+SIZE)) {
    printf("Стек переполнен.\n");
    exit(1);
  }
  *p1 = i;
}

int pop(void)
{
  if(p1==tos) {
    printf("Стек исчерпан.\n");
    exit(1);
  }
```

```
    p1--;
    return *(p1+1);
}
```

Как видим, стек реализован в виде массива **stack**. Сначала указатели **p1** и **tos** ссылаются на первый элемент стека. Затем указатель **p1** начинает перемещаться по стеку, а переменная **tos** по-прежнему хранит адрес его вершины. Это позволяет избежать переполнения стека и обращения к пустому стеку. Функции **push()** и **pop()** можно применять сразу после инициализации стека. В каждой из них выполняется проверка, не вышел ли указатель за пределы допустимого диапазона значений. В функции **push()** это позволяет предотвратить переполнение (значение указателя **p1** не должно превышать адрес **tos+SIZE**, где переменная **SIZE** определяет размер стека). В функции **pop()** эта проверка позволяет избежать обращения к пустому стеку (значение указателя **p1** не должно быть меньше указателя **tos**, ссылающегося на вершину стека).

Обратите внимание на то, что в функции **pop()** значение, возвращаемое оператором **return**, необходимо заключить в скобки. Без них оператор примет вид

```
return *p1 +1;
```

В этом случае возвращается значение, записанное по адресу **p1**, увеличенное на единицу, а не значение, хранящееся в ячейке с адресом **p1+1**.

Указатели и массивы

Указатели и массивы тесно связаны между собой. Рассмотрим следующий фрагмент программы.

```
char str[80], *p1;
p1 = str;
```

Здесь указателю **p1** присвоен адрес первого элемента массива **str**. Чтобы получить доступ к пятому элементу этого массива, следует выполнить один из двух операторов:

```
str[4]
```

или

```
*(p1+4)
```

Оба оператора вернут значение пятого элемента массива **str**. Напомним, что индексация массивов начинается с нуля, поэтому индекс пятого элемента массива **str** равен 4. Кроме того, пятый элемент массива можно получить, прибавив 4 к указателю **p1**, который вначале ссылается на первый элемент. (Как известно, имя массива без индекса представляет собой указатель на начало массива, т.е. на его первый элемент.)

Этот пример можно обобщить. По сути, в языке C/C++ существуют два способа обращения к элементам массива: индексация и адресная арифметика. Хотя индексация массива нагляднее, адресная арифметика иногда оказывается эффективнее. Поскольку быстродействие программы относится к одним из ее важнейших свойств, программисты на языке C/C++ широко используют указатели для обращения к элементам массива.

В следующем фрагменте программы приведены два варианта функции **putstr()**, иллюстрирующей доступ к элементам массива. В первом варианте используется индексация, а во втором — адресная арифметика. Функция **putstr()** предназначена для посимвольной записи строки в стандартный поток вывода.

```
/* Индексация указателя. */
void putstr(char *s)
{
  register int t;

  for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Доступ с помощью указателя. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

Большинство профессиональных программистов сочтут вторую версию более понятной и наглядной. На практике именно этот способ доступа к элементам массива распространен наиболее широко.

Массивы указателей

Как и все обычные переменные, указатели можно помещать в массив. Объявление массива, состоящего из 10 целочисленных указателей, выглядит следующим образом.

```
int *x[10];
```

Чтобы присвоить адрес целочисленной переменной **var** третьему элементу массива указателей, нужно выполнить оператор

```
x[2] = &var;
```

Чтобы извлечь значение переменной **var**, используя указатель **x[2]**, необходимо его разыменовать:

```
*x[2]
```

Массив указателей передается в функцию как обычно — достаточно указать его имя в качестве параметра. В следующем фрагменте на вход функции поступает массив указателей.

```
void display_array(int *q[])
{
  int t;

  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```

Напомним, что переменная **q** не является указателем на целочисленные переменные, ее следует интерпретировать как указатель на массив целочисленных указателей. Следовательно, необходимо объявить, что параметр **q** представляет собой массив целочисленных указателей. Это объявление сделано в заголовке функции.

Массивы часто содержат указатели на строки. Попробуем создать функцию, выводящую на экран сообщение об ошибке с заданным номером.

```
void syntax_error(int num)
{
  static char *err[] = {
    "Невозможно открыть файл\n",
    "Ошибка при чтении\n",
    "Ошибка при записи\n",
```

```
    "Отказ оборудования\n"
  };

  printf("%s", err[num]);
}
```

Указатели на строки содержатся в массиве **err**. Функция **printf()** вызывается внутри функции **syntax_error()** и выводит на экран строку с указанным номером. Например, если параметр **num** равен 2, на экране появится сообщение "Ошибка при записи".

Отметим, что аргумент командной строки **argv** также представляет собой массив указателей на символьные переменные (см. главу 3).

Косвенная адресация

Иногда указатель может ссылаться на другой указатель, который, в свою очередь, содержит адрес обычной переменной. Такая схема называется *косвенной адресацией* (multiple inderection), или *указателем на указатель* (pointer to pointer). Применение косвенной адресации снижает наглядность программы. Концепция косвенной адресации проиллюстрирована на рис. 5.3. Как видим, обычный указатель хранит адрес объекта, содержащего требуемое значение. В случае косвенной адресации один указатель ссылается на другой указатель, а тот, в свою очередь, содержит адрес объекта, в котором записано нужное значение.

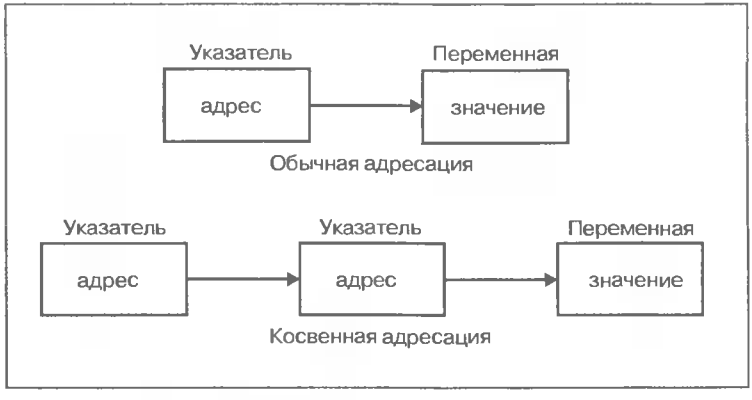


Рис. 5.3. Обычная и косвенная адресация

Глубина косвенной адресации не ограничена, однако почти всегда можно обойтись "указателем на указатель". Более громоздкие схемы адресации труднее понять, следовательно, вероятность ошибок при их использовании резко возрастает.

На заметку

Не путайте косвенную адресацию с такими динамическими структурами данных, как связанные списки, в которых используются указатели. Это принципиально разные понятия.

Переменная, представляющая собой указатель на указатель, объявляется следующим образом: перед ее именем записывается дополнительная звездочка. Например, в следующем операторе переменная **newbalance** объявляется как указатель на указатель на число с плавающей точкой.

```
float **newbalance;
```

Следует помнить, что переменная **newbalance** не является указателем на число с плавающей точкой, она лишь ссылается на указатель этого типа.

Чтобы извлечь значение переменной с помощью косвенной адресации, необходимо дважды применить оператор разыменования.

```
#include <stdio.h>

int main(void)
{
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* Вывод числа x */

  return 0;
}
```

Здесь переменная **p** объявлена как целочисленный указатель, а переменная **q** представляет собой указатель на целочисленный указатель. Функция **printf()** выводит на экран число 10.

Инициализация указателей

Если нестатический локальный указатель объявлен, но не инициализирован, его значение остается неопределенным. (Глобальные и статические локальные указатели автоматически инициализируются нулем.) При попытке применить указатель, содержащий неопределенный адрес, может разрушиться как программа, так и вся операционная система — хуже не придумаешь!

При работе с указателями большинство профессиональных программистов на языке C/C++ придерживаются следующего соглашения: указатель, не ссылающийся на конкретную ячейку памяти, должен быть равен нулю. По определению любой нулевой указатель ни на что не ссылается и не должен использоваться. Однако это еще не гарантирует безопасности. Использование нулевого указателя — всего лишь общепринятое соглашение. Это вовсе не правило, диктуемое языком C или C++. Например, если нулевой указатель поместить в левую часть оператора присваивания, риск разрушения программы и операционной системы остается высоким.

Поскольку предполагается, что нулевой указатель в вычислениях не используется, его можно применять для повышения наглядности и эффективности программ. Например, его можно использовать в качестве признака конца массива, содержащего указатели. Это позволит предотвратить выход за пределы допустимого диапазона. Подобная ситуация иллюстрируется на примере функции **search()**.

```
/* Поиск имени */
int search(char *p[], char *name)
{
  register int t;

  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;

    return -1; /* Имя не найдено */
}
```

Цикл **for** внутри функции **search()** выполняется до тех пор, пока не будет найдено требуемое имя, либо не обнаружится нулевой указатель. Как только нулевой указатель

будет обнаружен, условие цикла станет ложным, и программа передаст управление следующему оператору.

Программисты на C/C++ часто инициализируют строки. Пример такой инициализации приведен в функции **syntax_error()** из раздела "Массивы указателей". Рассмотрим еще один вариант инициализации строки в момент объявления.

```
char *p = "Здравствуйте";
```

Как видим, указатель **p** не является массивом. Попробуем понять, почему возможен такой способ инициализации. Все компиляторы языка C/C++ создают так называемую *таблицу строк* (string table), в которой хранятся строковые константы, используемые в программе. Следовательно, предыдущий оператор присвоит указателю **p** адрес строковой константы "Здравствуйте", записанной в таблицу строк. Указатель **p** используется в программе как обычная строка (однако изменять его нежелательно). Проиллюстрируем это следующим примером.

```
#include <stdio.h>
#include <string.h>

char *p = "Здравствуйте";

int main(void)
{
  register int t;

  /* Выводим строку в прямом и обратном порядке. */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);

  return 0;
}
```

С формальной точки зрения в стандарте языка C++ строковый литерал имеет тип **const char ***. Однако в языке C++ предусмотрено автоматическое преобразование втип **char ***. Таким образом, рассмотренная выше программа является абсолютно правильной. И все же эта особенность языка считается нежелательной, поэтому это преобразование не следует применять. Создавая новые программы, строковые литералы необходимо действительно считать константами, а объявление указателя **p** записывать следующим образом.

```
const char *p = "Здравствуйте";
```

Указатели на функции

Особенно малопонятным, хотя и действенным механизмом языка C++ являются *указатели на функции* (function pointer). Несмотря на то что функция не является переменной, она располагается в памяти, и, следовательно, ее адрес можно присваивать указателю. Этот адрес считается точкой входа в функцию. Именно он используется при ее вызове. Поскольку указатель может ссылаться на функцию, ее можно вызывать с помощью этого указателя. Это позволяет также передавать функции другим функциям в качестве аргументов.

Адрес функции задается ее именем, указанным без скобок и аргументов. Чтобы разобраться в этом механизме, рассмотрим следующую программу.

```
#include <stdio.h>
#include <string.h>
```

```
void check(char *a, char *b,
           int (*cmp)(const char *, const char *));

int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);

  p = strcmp;

  gets(s1);
  gets(s2);

  check(s1, s2, p);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Проверка равенства.\n");
  if(!(*cmp)(a, b)) printf("Равны");
  else printf("Не равны");
}
```

При вызове функция **check()** получает два указателя на символьные переменные и указатель на функцию. Соответствующие аргументы объявлены в ее заголовке. Обратите внимание на то, как объявлен указатель на функцию. Эту форму объявления следует применять для любых указателей на функции, независимо от того, какой тип имеют их аргументы и возвращаемые значения. Объявление ***cmp** заключено в скобки для того, чтобы компилятор правильно его интерпретировал.

Выражение

```
(*cmp)(a, b)
```

внутри функции **check()** означает вызов функции **strcmp()**, на которую ссылается указатель **cmp**, с аргументами **a** и **b**. Скобки по-прежнему необходимы. Это — один из способов вызвать функцию с помощью указателя. Альтернативный вызов выглядит так:

```
cmp(a, b);
```

Первый способ применяется чаще, поскольку он позволяет явно продемонстрировать, что функция вызывается через указатель. (Иначе говоря, читатель программы сразу увидит, что переменная **cmp** является указателем, а не именем функции.) Тем не менее эти два способа эквивалентны.

Обратите внимание на то, что функцию **check()** можно вызвать, непосредственно указав функцию **strcmp()** в качестве ее аргумента.

```
check(s1, s2, strcmp);
```

В этом случае отпадает необходимость в дополнительном указателе на функцию.

Может возникнуть закономерный вопрос: зачем вызывать функции с помощью указателей? На первый взгляд, это лишь усложняет программу, не предлагая взамен никакой компенсации. Тем не менее иногда выгоднее вызывать функции через указатели и даже создавать массивы указателей на функции. Рассмотрим в качестве примера синтаксический анализатор — составную часть компилятора, вычисляющую выражения. Он часто вызывает различные математические функции (синус, косинус, тан-

генс и т.д.), средства ввода-вывода или функции доступа к ресурсам системы. Вместо создания большого оператора **switch**, в котором пришлось бы перечислять все эти функции, можно создать массив указателей на них. В этом случае к функциям можно было бы обращаться по индексу. Чтобы оценить эффективность такого подхода, рассмотрим расширенную версию предыдущей программы. В этом примере функция **check()** проверяет на равенство строки, состоящие из букв или цифр. Для этого она просто вызывает разные функции, выполняющие сравнение.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check(char *a, char *b,
           int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b);

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  if(isalpha(*s1))
        check(s1, s2, strcmp);
  else
        check(s1, s2, numcmp);

  return 0;
}

void check(char *a, char *b,
           int (*cmp)(const char *, const char *))
{
  printf("Проверка равенства.\n");
  if(!(*cmp)(a, b)) printf("Равны");
  else printf("Не равны");
}

int numcmp(const char *a, const char *b)
{
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

Если пользователь введет строку, состоящую из букв, функция **check()** вызовет функцию **strcmp()**, получив указатель на нее. В противном случае будет передан указатель на функцию **numcmp()**. Итак, в разных ситуациях функция **check()** может вызывать разные функции.

Функции динамического расспределения памяти

Указатели позволяют динамически распределять память в программах на языке C/C++. Термин *динамическое распределение памяти* (dynamic allocation) означает, что программа может получать необходимую ей память уже в ходе своего выполнения.

Как известно, память для глобальных переменных выделяется на этапе компиляции. Локальные переменные хранятся в стеке. Следовательно, в ходе выполнения программы невозможно объявить новые глобальные или локальные переменные. И все же иногда в ходе выполнения программы возникает необходимость выделить дополнительную память, размер которой заранее не известен. Например, программа может использовать динамические структуры — связанные списки или деревья. Такие структуры данных являются динамическими по своей природе. Они увеличиваются или уменьшаются по мере необходимости. Для того чтобы реализовать такие структуры данных, программа должна иметь возможность распределять и освобождать память.

В языке C++ предусмотрены две системы для динамического распределения памяти: одна унаследована от языка C, вторая присуща лишь языку C++. Вторая система более совершенна. Ее мы рассмотрим во второй части книги, а пока сосредоточимся на системе динамического распределения памяти, предусмотренной языком C.

Память, выделяемая функциями динамического распределения, находится в *куче* (heap), которая представляет собой область свободной памяти, расположенную между кодом программы, сегментом данных и стеком. Хотя размер кучи заранее не известен, ее объем обычно достаточно велик.

В основе системы динамического распределения памяти в языке C лежат функции **malloc()** и **free()**X. (Во многих компиляторах предусмотрены дополнительные функции, позволяющие динамически выделять память, однако эти две — самые важные.) Эти функции создают и поддерживают список свободной памяти. Функция **malloc()** выделяет память для объектов, а функция **free()** освобождает ее. Иными словами, при каждом вызове функция **malloc()** выделяет дополнительный участок памяти, а функция **free()** возвращает его операционной системе. Любая программа, использующая эти функции, должна включать в себя заголовочный файл **stdlib.h**. (В программах на языке C++ можно также использовать новый стиль заголовочного файла **<cstdlib>**.)

Прототип функции **malloc()** имеет следующий вид.

void *malloc(size_t *количество_байтов*)

Параметр *количество_байтов* задает размер памяти, которую необходимо выделить. (Тип **size_t** определен в заголовочном файле **stdlib.h** как целое число без знака.) Функция **malloc()** возвращает указатель типа **void ***. Это означает, что его можно присваивать указателю любого типа. В случае успеха функция **malloc()** возвращает указатель на первый байт памяти, выделенной в куче. Если размера кучи недостаточно для успешного выделения памяти, функция **malloc()** возвращает нулевой указатель.

В приведенном ниже фрагменте программы выделяется 1000 байт непрерывной памяти.

```
char *p;
p = malloc(1000); /* get 1000 bytes */
```

После выполнения оператора присваивания указатель **p** ссылается на первый из 1000 байт выделенной памяти.

Обратите внимание на то, что в предыдущем примере указатель, возвращаемый функцией **malloc()**, присваивается указателю **p** без приведения типа. В языке C это допускается, поскольку указатель типа **void*** автоматически преобразовывается в тип указателя, стоящего в левой части оператора присваивания. Однако следует иметь в виду, что в языке C++ это правило *не действует*. Следовательно, в программах на языке C++ при присваивании указателя типа **void*** указателю другого типа необходимо выполнять явное приведение:

```
p = (char *) malloc(1000);
```

В принципе, это правило распространяется на любое присваивание или преобразование указателей. В этом заключается одно из принципиальных различий между языками C и C++.

В следующем примере выделяется память для 50 целых чисел. Обратите внимание на то, что применение оператора **sizeof** гарантирует машинонезависимость этого фрагмента программы.

```
int *p;
p = (int *) malloc(50*sizeof(int));
```

Поскольку размер кучи ограничен, при выделении памяти необходимо проверять указатель, возвращаемый функцией **malloc()**. Если он равен нулю, продолжать выполнение программы опасно. Проиллюстрируем эту мысль следующим примером.

```
p = (int *) malloc(100);
if(!p)
{
  printf("Память исчерпана.\n");
  exit(1);
}
```

Разумеется, если указатель **p** является нулевым, не обязательно выпрыгивать на ходу — вызов функции **exit()** можно заменить какой-нибудь обработкой возникшей ошибки и продолжить выполнение программы. Достаточно убедиться, что указатель **p** больше не равен нулю.

Функция **free()** является антиподом функции **malloc()** — она освобождает ранее занятую область динамической памяти. Освобожденную память можно снова использовать с помощью повторного вызова функции **malloc()**. Прототип функции **free()** выглядит следующим образом.

void free(void *p)

Здесь параметр *p* является указателем на участок памяти, ранее выделенный функцией **malloc()**. Принципиально важно то, что функцию **free()** ни в коем случае нельзя вызывать с неправильным аргументом — это разрушит список свободной памяти.

Проблемы, возникающие при работе с указателями

Указатели — головная боль программистов! С одной стороны, они представляют собой чрезвычайно мощный и необходимый во многих ситуациях механизм. С другой стороны, если указатель содержит непредвиденное значение, обнаружить эту ошибку крайне трудно.

Неверный указатель трудно найти, поскольку сам по себе он ошибкой не является. Проблемы возникают тогда, когда, пользуясь ошибочным указателем, вы начинаете считывать или записывать информацию в неизвестной области памяти. Если вы считываете ее, в худшем случае получите мусор, но если записываете — берегитесь! Вы можете повредить данные, код программы и даже операционную систему. Сначала ошибка может никак не проявиться, однако в ходе выполнения программы это может привести к непредсказуемым последствиям.[7] Такая ошибка напоминает мину замедленного действия, причем совершенно непонятно, где ее следует искать. Поистине, ошибочный указатель — это ночной кошмар программиста.

[7] Программа может даже успешно завершиться, но операционная система после ее выполнения может оказаться поврежденной. Таким образом, мало, чтобы программа делала то, что от нее требуют, нужно, чтобы она была безопасной для своего окружения. — *Прим. ред.*

Чтобы ваш сон был глубок и спокоен, при работе с указателями следует соблюдать особую осторожность. Рассмотрим несколько рецептов, позволяющих избежать ненужных проблем. Классический пример ошибки при работе с указателями — *неинициализированный указатель*. Рассмотрим следующую программу.

```
/* Осторожно: мины! */
int main(void)
{
  int x, *p;

  x = 10;
  *p = x;

  return 0;
}
```

В этой программе число 10 записывается в некую неизвестную область памяти. Причина заключается в том, что в момент присваивания `*p = x` указатель `p` содержит неопределенный адрес. Следовательно, невозможно предсказать, куда будет записано значение переменной `x`. Если программа невелика, такие ошибки могут себя никак не проявлять, поскольку вероятность того, что указатель `p` содержит "безопасный" адрес, достаточно велика, и ваши данные, программа и операционная система могут избежать опасности. Однако, если программа имеет большой размер, вероятность повредить жизненно важные области памяти резко возрастает. В конце концов программа перестанет работать. Для того чтобы избежать таких ситуаций, указатели всегда следует "заземлять", присваивая им корректные адреса.

Второй вид наиболее распространенных ошибок порождается простым неумением работать с указателями. Рассмотрим пример.

```
/* Осторожно — мины! */
#include <stdio.h>

int main(void)
{
  int x, *p;

  x = 10;
  p = x;

  printf("%d", *p);

  return 0;
}
```

При вызове функции `printf()` число 10 не будет выведено на экран. Вместо этого вы получите совершенно непредсказуемое значение, поскольку присваивание

```
p = x;
```

является неверным. Этот оператор присваивает число 10 указателю `p`. Однако указатель `p` должен содержать адрес, а не значение. Исправить программу можно следующим образом.

```
p = &x;
```

Еще одна разновидность ошибок вызывается неверными предположениями о размещении переменных в памяти компьютера. Программист не может заранее знать, где именно будут размещены его данные, окажутся ли они в том же месте при следующем запуске программы, и будут ли разные компиляторы обрабатывать их одина-

ково. Именно поэтому любые сравнения указателей, которые ссылаются на разные объекты, не имеют смысла. Рассмотрим пример.

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2) . . .
```

Этот фрагмент иллюстрирует типичную ошибку, связанную со сравнением указателей. (В очень редких ситуациях такие сравнения позволяют определить взаимное местоположение переменных. И все же это скорее исключение, чем правило.)

Аналогичные ошибки возникают, когда программист считает, будто два смежных массива можно индексировать одним указателем, пересекая границу между ними. Рассмотрим следующий фрагмент программы.

```
int first[10], second[10];
int *p, t;

p = first;
for(t=0; t<20; ++t)  *p++ = t;
```

Не стоит рисковать, инициализируя массивы подобным образом. Даже если в некоторых ситуациях этот способ сработает, нет никакой гарантии, что массивы **first** и **second** всегда будут располагаться в соседних ячейках памяти.

Следующая программа демонстрирует крайне опасную ошибку. Попробуйте сами найти ее.

```
/* Программа содержит ошибку! */
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  p1 = s;
  do {
    gets(s);  /* Считываем строку */

    /* Выводим десятичный эквивалент каждого символа */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "готово"));

  return 0;
}
```

В этой программе указатель **p1** используется для вывода на печать ASCII-кодов символов, содержащихся в строке **s**. Проблема заключается в том, что адрес строки **s** присваивается указателю **p1** только один раз, поэтому при втором проходе цикла он будет содержать адрес ячейки, на которой остановился в прошлый раз — ведь адрес начала строки ему не присваивается снова. В этой ячейке может быть записано все что угодно: символ из второй строки, другая переменная и даже инструкция программы! Исправить программу следует так.

```
/* Теперь все правильно! */
#include <string.h>
#include <stdio.h>

int main(void)
{
  char *p1;
  char s[80];

  do {
    p1 = s;
    gets(s);  /* Считываем строку */

   /* Выводим десятичный эквивалент каждого символа */
    while(*p1) printf(" %d", *p1++);

  } while(strcmp(s, "done"));

  return 0;
}
```

Теперь в начале каждой итерации указатель **p1** содержит адрес первого символа строки **s1**. Итак, при повторном использовании указателя его необходимо инициализировать вновь.

Неправильная работа с указателями может привести к тяжелым последствиям. Однако это не значит, что от указателей следует отказаться. Просто нужно быть внимательным и следить за тем, чтобы указатели содержали корректные адреса.

Полный справочник по

Глава 6

Функции

Функции — это строительные блоки для всех программ на языке C и C++. В этой главе мы рассмотрим функции, предусмотренные стандартом языка C, включая темы, связанные с передачей аргументов, возвращением значений, прототипами и рекурсией. Свойства функций, характерные для языка C++, мы рассмотрим во второй части книги. В частности, там будут освещены вопросы, связанные с перегрузкой функций и передачей параметров по ссылке.

Общий вид функции

Общий вид функции выглядит следующим образом.

```
тип_возвращаемого_значения имя_функции(список_параметров)
{
  тело функции
}
```

Тип_возвращаемого_значения определяет тип переменной, которую возвращает функция. Функция может возвращать переменные любого типа, кроме массива. В списке параметров перечисляются типы аргументов и их имена, разделенные запятыми. Если функция не имеет аргументов, ее *список_параметров* пуст. В этом случае скобки все равно необходимы.

В языке C/C++ с помощью одного оператора можно объявлять несколько переменных одинакового типа, разделяя их запятыми. В противоположность этому, каждый параметр функции должен быть объявлен отдельно. Таким образом, общий вид объявления параметров в заголовке функции выглядит так.

f(тип имя_переменной1, тип имя_переменной2, . . . , тип имя_переменнойN)

Рассмотрим примеры правильного и неправильного объявления параметров функций.

```
f(int i, int k, int j) /* Правильно */
f(int i, k, float j)   /* Неправильно */
```

Область видимости функции

Правила, определяющие область видимости (scope rules), устанавливают, видит ли одна часть программы другую часть кода или данных и может ли к ним обращаться.

Каждая функция представляет собой отдельный блок операторов. Код функции является закрытым и недоступным для любых операторов, расположенных в других функциях, кроме операторов вызова. (Например, невозможно перейти из одной функции в другую с помощью оператора `goto`.) Код, образующий тело функции, скрыт от остальной части программы, если он не использует глобальные переменные. Этот код не может влиять на другие функции, а они, в свою очередь, не могут влиять на него. Иначе говоря, код и данные, определенные внутри некой функции, никак не взаимодействуют с кодом и данными, определенными в другой функции, поскольку их области видимости не перекрываются.

Переменные, определенные внутри функции, называются *локальными* (local variables). Они создаются при входе в функцию и уничтожаются при выходе из нее. Иными словами, локальные переменные не сохраняют свои значения между вызовами функции. Единственное исключение из этого правила составляют статические локальные переменные. Они хранятся в памяти вместе с глобальными переменными, но

их область видимости ограничена функцией, в которой они определены. (Глобальные и локальные переменные подробно описаны в главе 2.)

В языке C (и C++) невозможно определить одну функцию внутри другой. По этой причине с формальной точки зрения ни язык C, ни язык C++ нельзя считать блочно-структурированными.

Аргументы функции

Если функция имеет аргументы, в ее заголовке должны быть объявлены переменные, принимающие их значения. Эти переменные называются *формальными параметрами* функции (formal parameters). Как и локальные переменные, они создаются при входе в функцию и уничтожаются при выходе из нее. Как показано в следующем примере, объявления параметров размещаются после имени функции.

```
/* Возвращает 1, если символ c является частью строки s,
   и 0, если нет. */
int is_in(char *s,  char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

Функция **is_in()** имеет два параметра: **s** и **c**. Она возвращает 1, если символ **c** является частью строки **s**, и 0, если нет.

Несмотря на то что эти переменные должны, в основном, получать значения аргументов, передаваемых функции, их можно использовать наравне с другими локальными переменными, в частности, присваивать им значения или использовать в любых выражениях.

Передача параметров по значению и по ссылке

В языках программирования существуют два способа передачи аргументов в подпрограмму. Первый из них известен как *передача параметров по значению* (call by value). В этом случае формальному параметру подпрограммы присваивается копия значения аргумента, и все изменения параметра никак не отражаются на аргументе.

Второй способ называется *передачей параметров по ссылке* (call by reference) . При использовании этого метода параметру присваивается адрес аргумента. Внутри подпрограммы этот адрес открывает доступ к фактическому аргументу. Это значит, что все изменения, которым подвергается параметр, отражаются на аргументе.

По умолчанию в языке C/C++ применяется передача по значению. Как правило, это означает, что код, образующий тело функции, не может изменять аргументы, указанные при ее вызове. Рассмотрим следующую программу.

```
#include <stdio.h>

int sqr(int x);

int main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);
```

```
  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

В этом примере значение аргумента функции **sqrt()**, т.е. число 10 копируется в параметр **x**. После выполнения присваивания **x = x * x** модифицируется только переменная **x**. Значение переменной **t**, указанной при вызове функции **sqr()**, по-прежнему равно 10.

Помните, что в функцию передается лишь копия аргумента. То, что произойдет с этой копией внутри функции, никак не отразится на оригинале.

Передача параметров по ссылке

Хотя в языке C/C++ по умолчанию применяется передача параметров по значению, их можно передавать и по ссылке. Для этого в функцию вместо аргумента нужно передать указатель на него. Поскольку функция получает адрес аргумента, ее код может изменять значение фактического аргумента вне функции.

Указатели передаются функции как обычные переменные. Естественно, они должны быть объявлены как указатели. Рассмотрим в качестве примера функцию **swap()**.

```
void swap(int *x, int *y)
{
  int temp;

  temp = *x;  /* Сохраняем значение, записанное по адресу x */
  *x = *y;    /* Записываем значение, записанное по адресу y,
                 в ячейку с адресом x */
  *y = temp;  /* Записываем исходное значение, записанное по
                 адресу x */
}
```

Функция **swap()** может менять местами значения двух переменных, на которые ссылаются указатели **x** и **y**, поскольку она получает их адреса (а не копии). Внутри этой функции содержимое переменных можно извлекать, используя обычные операции над указателями.

Помните, что функции **swap()** (как и любой другой функции, параметрами которой являются указатели) передаются *адреса аргументов*. Рассмотрим фрагмент, который демонстрирует правильный вызов функции **swap()**.

```
void swap(int *x, int *y);

int main(void)
{
  int i, j;

  i = 10;
  j = 20;

  swap(&i, &j); /* Передаются адреса переменных i и j */

  return 0;
}
```

В этом примере переменной **i** присваивается значение 10, а переменной **j** — число 20. Затем функции **swap()** передаются адреса переменных **i** и **j**, а не их значения. (Для этого используется унарный оператор получения адреса **&**.) После возврата управления из функции **swap()** переменные **x** и **y** оказываются переставленными.

На заметку *Язык C++ позволяет автоматически передавать параметры по ссылке, используя механизм ссылок, а не указатели. Это свойство описывается во второй части справочника.*

Передача массивов в качестве параметров

Массивы подробно описаны в главе 4. Однако в этом разделе мы рассмотрим передачу массивов в качестве параметров функций, поскольку они представляют исключение из обычного правила, в соответствии с которым в языке C/C++ все параметры должны передаваться по значению.

Если аргументом функции является массив, ей передается его адрес. Эта ситуация представляет собой исключение из общепринятого правила передачи параметров по значению. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать. Рассмотрим в качестве примера функцию **print_upper()**, предназначенную для преобразования строчных букв в прописные (с кириллицей программа не работает — *Прим. ред.*).

```
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("\nСтрока из прописных букв: %s", s);
  return 0;
}

/* Выводит строку, состоящую из прописных букв. */
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)  {
    string[t] = toupper(string[t]);
    putchar(string[t]);
  }
}
```

После вызова функции **print_upper()** буквы, хранящиеся в массиве **s**, заменяются прописными. Если вы не хотите изменять содержимое массива **s**, перепишите программу следующим образом.

```
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);
```

```
int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("\nСтрока не изменилась: %s", s);

  return 0;
}

void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)
    putchar(toupper(string[t]));
}
```

В этой версии программы содержимое массива **s** остается постоянным, поскольку его значение в функции **print_upper()** не изменяется.

Стандартная библиотечная функция **gets()** представляет собой классический пример передачи массива в качестве параметра. Рассмотрим ее аналог **xgets()**, хотя реальная функция **gets()** намного сложнее.

```
/* Упрощенный вариант стандартной библиотечной
   функции gets(). */
char *xgets(char *s)
{
  char ch, *p;
  int t;

  p = s;  /* Функция xgets() возвращает указатель на строку s */

  for(t=0; t<80; ++t){
    ch = getchar();

    switch(ch) {
      case '\n':
        s[t] = '\0'; /* Признак конца строки */
        return p;
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
  }
  s[79] = '\0';
  return p;
}
```

Аргументом функции **xgets()** должен быть указатель на символьную переменную. Разумеется, в качестве аргумента можно задать имя символьного массива, которое по определению является указателем типа **char ***. Сначала функция **xgets()** выполняет цикл **for**, счетчик которого изменяется с 0 до 79. Это предотвращает ввод с клавиатуры более длинной строки. Если пользователь попытается ввести больше 80 символов, функция выполнит оператор **return**. (В реальной функции **gets()** этого ограничения

нет.) Поскольку в языке C/C++ нет встроенной проверки выхода индекса массива за пределы допустимого диапазона, следует убедиться, что любой массив, передаваемый функции **xgets()**, может вместить до 80 символов. Каждый символ, введенный с клавиатуры, помещается в строку. При нажатии клавиши **<BACKSPACE>** счетчик **t** уменьшается на 1, удаляя из массива последний символ. При нажатии клавиши **<ENTER>** в конец строки помещается нулевой байт. Поскольку функция **xgets()** модифицирует фактический аргумент, после выхода из нее массив **s** будет содержать символы, введенные пользователем.

Аргументы функции main(): argc и argv

Иногда бывает удобно вводить данные, указывая их в командной строке при запуске программы. Для этого обычно используются аргументы функции **main()**. Аргументами командной строки называются данные, указанные в командной строке операционной системы вслед за именем программы. Например, при компиляции программы часто используют командную строку

cc имя_программы

Здесь параметр *имя_программы* является аргументом командной строки, который задает имя компилируемой программы.

Для получения аргументов командной строки предназначены встроенные аргументы функции **main()**: **argv** и **argc**. Целочисленный параметр **argc** содержит количество аргументов командной строки. Его значение не может быть меньше 1, поскольку имя программы считается первым аргументом. Параметр **argv** представляет собой указатель на массив символьных указателей. Каждый элемент этого массива ссылается на аргумент командной строки. Все аргументы командной строки являются строками — введенные числа должны конвертироваться в соответствующее внутреннее представление. Рассмотрим простую программу, которая выводит на экран строку "Привет," и ваше имя, введенное в командной строке.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Вы забыли указать свое имя.\n");
    exit(1);
  }
  printf("Привет, %s!", argv[1]);

  return 0;
}
```

Если в командной строке набрать имя программы и указать свое имя, например, **name Иван**, то программа выведет на экран строку "Привет, Иван!".

Во многих операционных системах каждый аргумент командной строки должен отделяться пробелом или знаком табуляции. Запятые, точки с запятой и другие знаки пунктуации разделителями не считаются. Рассмотрим пример.

```
run Spot, run
```

Эта командная строка состоит из трех строк. А командная строка

```
Herb,Rick,Fred
```

содержит только одну строку, поскольку запятая разделителем не является.

В некоторых операционных системах строку с пробелами можно заключать в двойные кавычки. В этом случае она считается отдельным аргументом. Прежде чем передавать параметры через командную строку, прочитайте документацию, сопровождающую вашу операционную систему.

Аргумент **argv** необходимо объявлять самому. Чаще всего применяется следующий способ.

```
char *argv[];
```

Пустые квадратные скобки означают, что массив имеет неопределенную длину. Доступ к отдельным элементам массива **argv** осуществляется с помощью индексации. Например, элемент **argv[0]** ссылается на первую строку, которая всегда является именем программы, элемент **argv[1]** ссылается на первый аргумент и т.д.

Рассмотрим программу **countdown**, в которой используются аргументы командной строки. Счетчик цикла в этой программе уменьшается с начального значения, заданного аргументом командной сроки, до 0. Обратите внимание на то, что первый аргумент преобразуется в целое число с помощью стандартной функции **atoi()**. Если вторым аргументом является строка "display", значение счетчика выводится на экран.

```
/* Программа countdown. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int disp, count;

  if(argc<2) {
    printf("Введите начальное значение счетчика\n");
    printf("в командной строке.  Попробуйте еще.\n");
    exit(1);
  )

  if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
  else disp = 0;

  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

  putchar('\a');  /* Сигнал */
  printf("Готово");

  return 0;
}
```

Если командная строка не содержит аргументов, на экран выдается сообщение об ошибке. Программа, ожидающая ввода аргументов из командной строки, должна анализировать их и давать пользователю соответствующие инструкции.

Чтобы выделить отдельный символ из аргумента командной строки, необходимо использовать индекс массива **argv**. Следующая программа выводит на экран все свои аргументы посимвольно.

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
  int t, i;

  for(t=0; t<argc; ++t) {
    i = 0;

    while(argv[t][i]) {
      putchar(argv[t][i]);
      ++i;
    }
    printf("\n");
  }

  return 0;
}
```

Учтите, что первый индекс предоставляет доступ к строке, а второй — к отдельному символу.

Обычно аргументы **argc** и **argv** используются для передачи начальных параметров программы. Теоретически можно указывать до 32767 аргументов, однако в большинстве операционных систем это количество намного меньше. Как правило, аргументы командной строки задают имя файла или какую-либо опцию. Применение командных аргументов придает программе профессиональный вид и позволяет применять пакетные файлы (batch files).

Если программа не использует аргументов, указываемых в командной строке, как правило, функция **main()** явно объявляется без параметров. В программах на языке C для этого достаточно просто указать в списке параметров ключевое слово **void**. Однако в языке C++ список параметров можно вообще оставить пустым. Использование ключевого слова **void** для объявления пустого списка параметров в языке C++ допускается, но не рекомендуется.

Несмотря на то что имена **argc** и **argv** являются традиционными, аргументы функции **main()** можно называть как угодно. Кроме того, компиляторы могут поддерживать расширенный список аргументов функции **main()**. Информацию об этом следует искать в документации.

Оператор return

Оператор **return** уже рассматривался в главе 3. Как известно, он используется в двух ситуациях. Во-первых, он вызывает немедленный выход из функции. Иначе говоря, программа передает управление вызывающему модулю. Во-вторых, с помощью оператора **return** осуществляется возврат значения, вычисленного функцией. Рассмотрим, как это происходит.

Возврат управления из функции

Существует два способа прекратить выполнение функции и вернуть управление вызывающему модулю. В первом случае функция полностью выполняет свои операторы, и управление достигает закрывающей фигурной скобки **}**. (Разумеется, сама фигурная скобка не порождает никакого объектного кода, однако ее можно так интерпретировать.) Например, функция **pr_reverse()** в следующей программе выводит на экран строку "Я люблю C++" в обратном порядке, а затем возвращает управление главному модулю.

```
#include <string.h>
#include <stdio.h>

void pr_reverse(char *s);

int main(void)
{
  pr_reverse("Я люблю C++");

  return 0;
}

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Как только строка появляется на экране, функции **pr_reverse()** ничего не остается делать, и она возвращает управление в точку, откуда была вызвана.

На самом деле этот способ используется довольно редко, поскольку, во-первых, это снижает наглядность и эффективность функции, а во-вторых, в большинстве случаев она должна вернуть некий результат.

Функция может содержать несколько операторов **return**. Например, функция **find_substr()** в следующей программе возвращает начальную позицию подстроки, входящей в заданную строку, а если подстрока не найдена, возвращает число -1.

```
#include <stdio.h>

int find_substr(char *s1, char *s2);

int main(void)
{
  if(find_substr("C++ прекрасен", "красен") != -1)
    printf("подстрока не найдена");

  return 0;
}

/* Возвращает индекс первого символа подстроки s2,
   входящей в строку s1. */
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

  for(t=0; s1[t]; t++) {
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* Первый оператор return */
  }
   return -1; /* Второй оператор return */
}
```

Возвращаемые значения

Все функции, за исключением функций, объявленных со спецификатором **void**, с помощью оператора **return** возвращают некий результат. В соответствии со стандартом C89, если функция, возвращающая значение, не использует для этого оператор **return**, в вызывающий модуль возвращается "мусор", т.е. случайное значение. В языке C++ (и в стандарте C99) все функции, возвращающие значение, *должны* делать это с помощью оператора **return**. Иными словами, если в объявлении указано, что функция возвращает некое значение, любой оператор **return**, расположенный в ее теле, должен быть связан с определенной переменной. Однако, если поток управления достигнет конца функции, возвращающей значение, не встретив на своем пути оператор **return**, в вызывающий модуль также вернется "мусор". Несмотря на то что эта ошибка не относится к разряду синтаксических, она является принципиальным изъяном программы и должна быть исключена.

Если функция не объявлена со спецификатором **void**, ее можно использовать в качестве операнда в любом выражении. Таким образом, фрагмент программы, приведенный ниже, является совершенно правильным.

```
x = power(y);
if(max(x,y) > 100) printf("больше");
for(ch=getchar(); isdigit(ch); ) ... ;
```

Как правило, функция не должна стоять в левой части оператора присваивания. Следовательно, оператор

```
swap(x,y) = 100; /* Неправильный оператор */
```

является ошибочным. Компилятор языка C/C++ распознает эту ошибку и выдаст соответствующее сообщение. (Как указывается в части 2, язык C++ допускает несколько интересных исключений из этого правила, что позволяет указывать некоторые функции в левой части оператора присваивания.)

Функции можно разделить на три категории. К первой категории относятся вычислительные функции. Они выполняют некие операции над своими аргументами и возвращают результат вычислений. Их можно назвать "настоящими" функциями. К примеру, "настоящими" являются стандартные библиотечные функции **sqrt()** и **sin()**, вычисляющие квадратный корень и синус соответственно.

Функции второго типа выполняют обработку информации. Их возвращаемое значение просто означает, успешно ли завершены операции. В качестве примера можно указать библиотечную функцию **fclose()**, закрывающую файл. Если файл закрыт успешно, функция возвращает значение 0, в противном случае она возвращает константу **EOF**.

И, наконец, функции третьего вида не имеют явно возвращаемого значения. По существу, эти функции являются процедурами и не вычисляют никакого результата. К этой категории относится функция **exit()**, прекращающая выполнение программы. Все функции, не возвращающие никаких значений, следует объявлять с помощью спецификатора **void**. Такие функции нельзя использовать в выражениях.

Иногда функции, не вычисляющие ничего интересного, все равно что-то возвращают. Например, функция **printf()** возвращает количество символов, выведенных на экран. Было бы интересно увидеть программу, в которой это значение имело какой-то смысл. Иными словами, хотя все функции, за исключением функций, объявленных с помощью спецификатора **void**, возвращают какие-то значения, использовать их в дальнейших вычислениях совершенно не обязательно. При обсуждении функций часто возникает вопрос: "Обязательно ли присваивать значение, возвращаемое функцией, какой-то переменной, объявленной в вызывающем модуле?". Нет, не обязательно. Если оператор присваивания не указан, возвращаемое значение просто игнорируется. Рассмотрим программу, в которой используется функция **mul()**.

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
  int x, y, z;

  x = 10;   y = 20;
  z = mul(x, y);           /* 1 */
  printf("%d", mul(x,y));  /* 2 */
  mul(x, y);               /* 3 */

  return 0;
}

int mul(int a, int b)
{
  return a*b;
}
```

В первой строке значение, возвращаемое функцией **mul()**, присваивается переменной **z**. Во второй строке возвращаемое значение ничему не присваивается, но используется внутри функции **printf()**. И, наконец, в третьей строке значение, возвращаемое с помощью оператора **return**, отбрасывается, поскольку оно ничему не присваивается и не является частью какого-либо выражения.

Возврат указателей

Несмотря на то что функции, возвращающие указатели, ничем не отличаются от обычных, с ними связано несколько важных понятий.

Указатели не являются целочисленными переменными. Они содержат адреса переменных, имеющих определенный тип. Используя адресную арифметику, следует иметь в виду, что результаты ее операций зависят от базового типа указателей. Например, при увеличении целочисленного указателя на единицу его значение увеличится на 4 (если целые числа занимают 4 байт памяти). Как правило, при каждом увеличении (или уменьшении) указателя на единицу, он перемещается на следующую (или предыдущую) переменную этого типа. Поскольку размер переменных, имеющих разные типы, варьируется, компилятор должен знать, с каким типом данных связан тот или иной указатель. По этой причине функция, возвращающая указатель, должна явно объявлять его тип. Например, нельзя применять оператор **return** для возврата указателя, имеющего тип **int ***, если в объявлении функции указано, что она возвращает указатель типа **char ***!

Чтобы вернуть указатель, в объявлении функции следует указать соответствующий тип возвращаемого значения. Например, следующая функция возвращает указатель на первое вхождение символа **c** в строку **s**.

```
/* Возвращает указатель на первое вхождение
   символа c в строку s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

Если символ не найден, возвращается нуль. Рассмотрим короткую программу, в которой используется функция **match()**.

```
#include <stdio.h>

char *match(char c, char *s);  /* Прототип */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p)  /* Есть вхождение */
    printf("%s ", p);
  else
    printf("Символ не обнаружен.");

  return 0;
}
```

Эта программа сначала считывает строку, а затем — символ. Если символ входит в строку, программа выводит ее на экран, начиная с позиции, в которой обнаружен искомый символ. В противном случае программа выдает сообщение "Символ не обнаружен".

Функции типа void

Функции, объявленные с помощью спецификатора **void**, не возвращают никаких значений. Это не позволяет использовать такие функции в выражениях и предотвращает их неправильное применение. Например, функция **print_vertical()** выводит на экран строку, переданную ей в качестве аргумента, размещая ее символы один под другим. Поскольку эта функция ничего не возвращает, на месте типа возвращаемого значения указано ключевое слово **void**.

```
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Рассмотрим пример, в котором используется функция **print_vertical()**.

```
#include <stdio.h>

void print_vertical(char *str);  /* Прототип */

int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}

void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

И последнее, в ранних версиях языка C ключевое слово **void** не определялось. Таким образом, в старых программах функции, не возвращающие никаких значений, вместо спецификатора **void** по умолчанию использовали тип **int**. Не удивляйтесь, встретив такие примеры где-нибудь в архиве.

Зачем нужен оператор return в функции main()

Функция **main()** возвращает целое число вызвавшему ее процессу, как правило, операционной системе. Вызвав функцию **exit()** с этим аргументом, вы получите тот же самый эффект. С формальной точки зрения, если функция **main()** ничего не возвращает явным образом, то вызывающему процессу передается неопределенное значение. На практике многие компиляторы языка C/C++ по умолчанию возвращают число 0, но полагаться на это не стоит, поскольку это снизит машинонезависимость вашей программы.

Рекурсия

В языке C/C++ функция может вызывать саму себя. Функции, вызывающие сами себя, называются *рекурсивными* (recursive). Рекурсия — это процесс определения какого-либо понятия через него же. Иногда его называют *круговым определением* (circular definition).

Простым примером рекурсивной функции является функция **factr()**, вычисляющая факториал целого числа. Факториалом называется число **n**, представляющее собой произведение всех целых чисел от 1 до **n**. Например, факториал числа 3 равен $1 \times 2 \times 3 = 6$. Рассмотрим рекурсивный и итеративный варианты функции **factr()**.

```
/* Рекурсивный вариант */
int factr(int n)
{
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n; /* рекурсивный вызов */
  return(answer);
}

/* Итеративный вариант */
int fact(int n) {
  int t, answer;

  answer = 1;

  for(t=1; t<=n; t++)
    answer=answer*(t);

  return(answer);
}
```

Итеративный вариант функции **factr()** выглядит проще. В нем используется цикл, счетчик которого пробегает значения от 1 до **n** и последовательно умножается на предыдущий результат.

Рекурсивная версия функции **factr()** несколько сложнее. Если функция **factr()** вызывается с аргументом 1, она возвращает число 1. В противном случае она возвращает значение **factr(n-1)*n**. Чтобы вычислить это выражение, функция **factr()** вызывается **n-1** раз до тех пор, пока значение переменной **n** не станет равным 1.

В этот момент начнется последовательное выполнение операторов `return`, относящихся к разным вызовам функции `factr()`.

При вычислении факториала числа 2 первый вызов функции `factr()` порождает второй вызов этой функции — теперь уже с аргументом, равным 1. Второй вызов вернет число 1, которое будет умножено на 2 (исходное значение аргумента `n`). Попробуйте сами проследить за вызовами функции `factr()` при вычислении факториала числа 3. (Чтобы увидеть уровень каждого вызова функции `factr()`, можно вставить в нее вызов функции `printf()` и вывести промежуточные результаты.)

Когда функция вызывает саму себя, в стеке размещается новый набор ее локальных переменных и параметров, и функция выполняется с начала. Рекурсивный вызов не создает новую копию функции. Копируются лишь переменные, с которыми она работает. При каждом рекурсивном возврате управления копии переменных и параметров удаляются из стека, а выполнение программы возобновляется с места вызова функции. Рекурсивные функции напоминают телескоп, который то раскладывается, то складывается.

Как правило, рекурсивные функции незначительно уменьшают размер кода и ненамного повышают эффективность использования памяти по сравнению с ее итеративными аналогами. Кроме того, рекурсивные версии большинства функций выполняются несколько медленнее, чем их итеративные эквиваленты. Большое количество рекурсивных вызовов может вызвать переполнение стека, поскольку при каждом вызове функции в стеке размещается новый набор ее локальных переменных и параметров. При этом программист может пребывать в полном неведении, пока не произойдет аварийное прекращение работы программы.

Основное преимущество рекурсивных функций заключается в том, что они упрощают и делают нагляднее некоторые алгоритмы. Например, алгоритм быстрой сортировки трудно реализовать итеративным способом. Кроме того, некоторые проблемы, связанные с искусственным интеллектом, легче решить, применяя рекурсию. В заключение отметим, что некоторым людям легче думать рекурсивно, чем итеративно.

Создавая рекурсивную функцию, необходимо предусмотреть условный оператор, например, оператор `if`, с помощью которого можно прекратить выполнение рекурсивных вызовов. Если этого не сделать, функция может бесконечно вызывать саму себя. Отсутствие условия остановки рекурсии является весьма распространенной ошибкой. Для отслеживания промежуточных результатов следует широко применять функцию `printf()`. Это позволит сохранять контроль за вычислениями и прервать выполнение программы, если возникла ошибка.

Прототипы функций

В языке C++ все функции должны быть объявлены до своего первого вызова. Обычно для этого используются *прототипы функций* (function prototypes). В исходном варианте языка C прототипов не было. Они были добавлены при разработке стандарта, в котором настоятельно рекомендуется использовать прототипы. Следует заметить, что это лишь пожелание, а не категорическое требование. В то же время, в языке C++ прототипы являются *обязательными*. В этой книге все функции имеют полные прототипы. Это позволяет выполнять более строгую проверку типов, примерно так же, как в языке Pascal. Если в программе используются прототипы, компилятор может обнаружить несоответствия между типами аргументов и параметрами функций, а также несовпадение их количества.

Общий вид прототипа таков.

тип имя_функции(*тип имя_параметра1*, *тип имя_параметра2*, ...,
тип имя_параметраN)

Указывать имена параметров не обязательно. Однако они дают компилятору возможность обнаружить имя параметра, тип которого не совпадает с именем аргумента, поэтому рекомендуется использовать имена аргументов в прототипах.

Следующая программа иллюстрирует ценность прототипов. При компиляции вы получите сообщение об ошибке, поскольку программа содержит вызов функции **sqr_it()** с целочисленным аргументом, в то время как ей следует передать указатель целого типа. (Преобразование целочисленной переменной в указатель не допускается.)

```
/* Прототип функции обеспечивает строгую проверку типов. */

void sqr_it(int *i); /* Прототип */

int main(void)
{
  int x;

  x = 10;
  sqr_it(x);  /* Несовпадение типов */

  return 0;
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

Если определение функции размещено до ее первого вызова, оно само считается прототипом. Таким образом, программа, приведенная ниже, является совершенно правильной.

```
#include <stdio.h>

/* Это определение функции служит ее прототипом. */
void f(int a, int b)
{
  printf("%d ", a % b);
}

int main(void)
{
  f(10,3);

  return 0;
}
```

Поскольку функция **f()** определена до ее вызова в функции **main()**, отдельный прототип не нужен. Объединение определения с прототипом чаще всего используют в маленьких программах. В больших программах такой прием встречается редко, особенно если программа состоит из нескольких файлов. Все программы, приведенные в нашей книге, содержат отдельные прототипы функций, поскольку именно такой стиль считается правильным.

Единственная функция, для которой не требуется прототип, — функция **main()**, поскольку она всегда вызывается первой.

Следует учитывать небольшую, но важную разницу между прототипами функций, не имеющих параметров, в языках C и C++. В языке C++ пустой список параметров просто обозначается пустыми скобками. Например, так.

```
int f(); /* Прототип функции, не имеющей параметров,
            в языке C++ */
```

Однако в языке C этот прототип трактуется иначе. По историческим причинам пустой список параметров в языке C означает *отсутствие информации о параметрах*. В зависимости от компилятора это может означать, что функция либо не имеет параметров, либо имеет несколько параметров. В языке C прототип функции, не имеющей параметров, должен содержать ключевое слово **void**. Вот как должен выглядеть прототип функции **f()** в программе на языке C.

```
float f(void);
```

Это объявление сообщает компилятору, что функция не имеет параметров, и любой вызов этой функции с какими-либо параметрами является ошибкой. В языке C++ такой способ объявления пустого списка параметров также допускается, но считается излишним.

Внимание! *В языке C++ выражения* **f()** *и* **f(void)** *эквивалентны.*

Прототипы функций позволяют выявлять ошибки в программе. Кроме того, они гарантируют, что функция не будет вызвана с неверными параметрами.

И последнее, поскольку в ранних версиях языка C прототипы не поддерживались, их применение в программах на C является необязательным. При переводе программ с языка C на язык C++ следует добавить полные прототипы всех функций. Помните, несмотря на то что прототипы в языке C не обязательны, в языке C++ они необходимы. Следовательно, любая функция в программе на языке C++ должна иметь полный прототип.

Прототипы стандартных библиотечных функций

Программа должна содержать прототипы всех вызываемых стандартных библиотечных функций. Для этого в программу следует включить соответствующие *заголовочные файлы*, поддерживаемые компиляторами языка C/C++. В языке C заголовочные файлы обычно имеют расширения **.h**. В языке C++ заголовочные файлы могут быть как отдельными, так и создаваться самим компилятором. В любом случае, заголовочный файл состоит из двух элементов: определений переменных и функций, используемых в библиотечных функциях, а также прототипов самих библиотечных функций. Например, заголовочный файл **stdio.h** включается почти во все программы, рассмотренные в книге, поскольку он содержит прототип функции **printf()**. Заголовочные файлы для стандартной библиотеки описаны в части III.

Определение списка параметров переменной длины

В программе можно объявить функцию с переменным количеством параметров. В большинстве случаев это относится к функции **printf()**. Чтобы сообщить компилятору, что функция может быть вызвана с разным количеством параметров, объявление ее параметров следует завершить многоточием. Например, прототип, приведенный ниже, означает, что функция **func()** может иметь по меньшей мере два параметра и сколько угодно дополнительных параметров (или вовсе не иметь их).

```
int func(int a, int b, ...);
```

Этот способ объявления параметров используется и в определении функции.

Любая функция с переменным количеством параметров должна иметь по меньшей мере один фактический параметр. Например, следующее объявление является неправильным.

```
int func(...); /* Неправильно */
```

Объявление параметров функции в классическом и современном стиле

Объявление параметров функции в ранних версиях языка С отличается от стандартного. Иногда этот способ называют *классическим*. В нашей книге мы используем *современный* стиль. Стандарт языка С допускает оба стиля, но настоятельно рекомендует использовать лишь современный. В языке С++ поддерживается только современный стиль объявления параметров функции. Однако следует помнить о существовании старого стиля, поскольку многие программы, написанные на старом языке С, используются до сих пор.

Классическое объявление параметров функции состоит из двух частей: списка параметров, заключенного в скобки и указанного вслед за именем функции, и объявлений фактических параметров, расположенных между заголовком и телом функции.

```
тип имя_функции(параметр1, параметр2, ..., параметрN)
тип параметр1;
тип параметр2;
.
.
.
тип параметрN;
{
  тело функции
}
```

Например, современное объявление параметров

```
float f(int a, int b, char ch)
{
  /* ... */
}
```

в классическом варианте выглядит так:

```
float f(a, b, ch)
int a, b;
char ch;
{
  /* ... */
}
```

Обратите внимание на то, что классическое определение позволяет объявлять несколько параметров одновременно.

Внимание! *Классический стиль объявления параметров объявлен устаревшим и не поддерживается языком С++.*

Полный справочник по

Глава 7

Структуры, объединения, перечисления и оператор typedef

В языке C существуют пять способов создать пользовательский тип.

1. *Структура* (structure), позволяющая объединить группу переменных под одним именем. Структура часто называется *агрегированным* типом данных. (Распространены также термины *составной*, или *совокупный*, тип.)
2. *Битовое поле* (bit-field), представляющее собой вариант структуры и позволяющее работать с отдельными битами.
3. *Объединение* (union), позволяющее использовать одну и ту же область памяти для хранения нескольких переменных разного типа.
4. *Перечисление* (enumeration), представляющее собой список именованных целочисленных констант.
5. Ключевое слово **typedef**, дающее возможность присваивать существующему типу новое имя.

В языке C++ поддерживаются все пять способов и добавляются классы, описанные в части II.

На заметку *В языке C++ структуры и объединения могут иметь как объектно-ориентированные, так и процедурные свойства. В этой главе описываются лишь императивные особенности этих типов. Их объектно-ориентированные свойства будут рассмотрены позднее.*

Структуры

Структура — это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных. *Объявление структуры* создает ее шаблон, который можно использовать при создании объектов структуры (т.е. ее экземпляров). Переменные, входящие в структуру, называются ее *членами* (members). (Члены структуры часто называют также ее *элементами* или *полями*.)

Как правило, все члены структуры логически связаны друг с другом. Например, имя и адрес в списке рассылки естественно представлять в виде структуры. Следующий фрагмент программы демонстрирует, как объявляется структура, состоящая из полей, в которых хранятся имена и адреса. Ключевое слово **struct** сообщает компилятору, что объявляется именно структура.

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

Обратите внимание на то, что объявление завершается точкой с запятой. Это необходимо, поскольку объявление структуры является оператором. Данная структура имеет тип **addr**. Таким образом, тип **addr** идентифицирует конкретную структуру данных и является ее спецификатором.

В приведенном выше фрагменте *еще не создана ни одна переменная*. В нем лишь определен составной тип данных, а не сама переменная. Для того чтобы возникла реальная переменная данного типа, ее нужно объявить отдельно. В языке C переменная типа **addr** (т.е. физический объект в памяти компьютера) создается оператором

```
struct addr addr_info;
```

Здесь объявляется переменная типа **addr** с именем **addr_info**. В языке C++ можно использовать более короткую форму записи.

```
addr addr_info;
```

Как видим, ключевое слово **struct** здесь не понадобилось. В языке C++ принято правило: после объявления структуры переменные этого типа можно объявлять, указывая лишь их тип, и ставить перед ним ключевое слово **struct** не обязательно. Причина заключается в том, что в языке C имя структуры не определяет полное имя типа. Фактически в стандарте языка C имя структуры считается *дескриптором* (tag). В то же время в языке C++ имя структуры полностью определяет ее тип, поэтому его можно использовать при определении переменных. Однако следует иметь в виду, что стиль объявления структуры вполне допустим в программах на языке C++. Поскольку программы, приведенные в части I, соответствуют стандартам обоих языков, мы используем способ, принятый в языке C. Просто помните, что в языке C++ существует более короткая форма записи.

После объявления переменной, представляющей собой структуру (например, переменной **addr_inf**), компилятор автоматически выделяет память для ее членов. На рис. 7.1 показана схема размещения в памяти переменной **addr_info** с учетом того, что символы занимают 1 байт, а целые числа — 4.

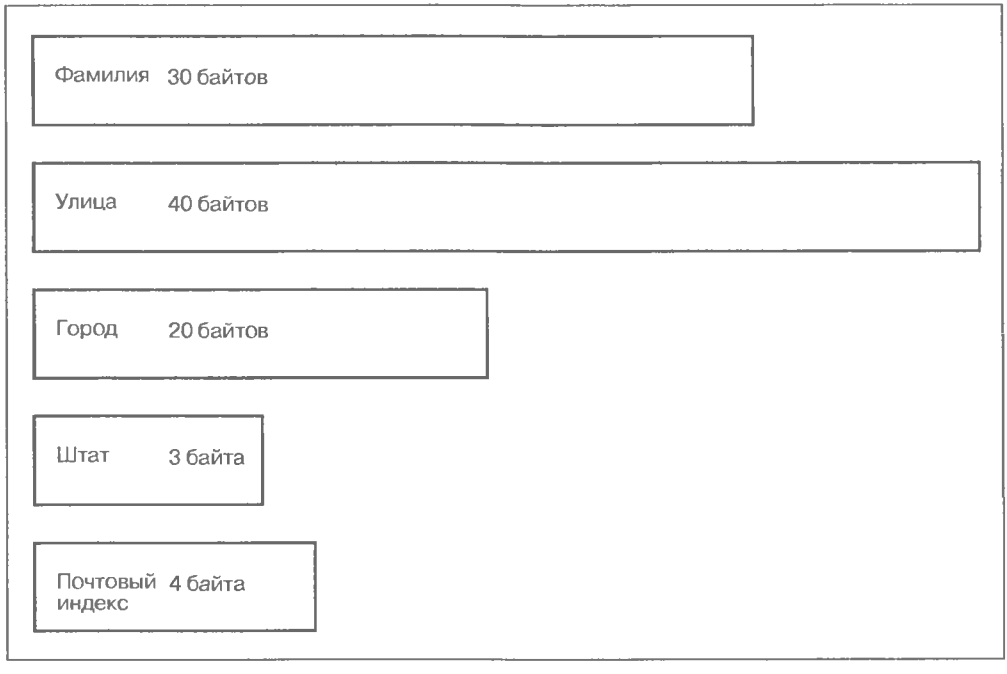


Рис. 7.1. Схема расположения структуры addr_info в памяти

Одновременно с определением структуры можно объявить несколько ее экземпляров. Например, в операторе

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info, binfo, cinfo;
```

определяется структура **addr** и объявляются переменные **addr_info**, **binfo** и **cinfo**, являющиеся ее экземплярами. Важно понимать, что каждый экземпляр структуры содержит свои собственные копии членов структуры. Например, поле **zip** в объекте **binfo** отличается от поля **zip**, принадлежащего переменной **cinfo**. Таким образом, изменения поля **zip**, относящегося к объекту **binfo**, никак не повлияют на поле **zip** в переменной **cinfo**.

Если в программе нужен лишь один экземпляр структуры, ее имя указывать не обязательно. Иначе говоря, оператор

```
struct {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

объявляет одну переменную с именем **addr_info**, представляющую собой экземпляр структуры, определенной выше.

Объявление структуры имеет следующий общий вид.

```
struct имя_типа_структуры {
  тип имя_члена;
  тип имя_члена;
  тип имя_члена;
  .
  .
  .
} имена_экземпляров;
```

Здесь *имя_типа_структуры* и *имена_экземпляров* можно не указывать, правда, не одновременно.

Доступ к членам структуры

Доступ к отдельным членам структуры обеспечивается оператором "**.**" (обычно его называют *оператором "точка"* или *оператором доступа к члену структуры*). Например, в следующем фрагменте программы полю **zip** структуры **addr_info**, объявленной ранее, присваивается почтовый индекс 12345.

```
addr_info.zip = 12345;
```

Имя экземпляра структуры указывается перед точкой, а имя члена структуры — после нее. Оператор, предоставляющий доступ к члену структуры, имеет следующий вид:

имя_экземпляра.имя_члена

Таким образом, чтобы вывести на экран индекс, нужно выполнить следующий оператор:

```
printf("%d", addr_info.zip);
```

В результате на экране появится почтовый индекс, содержащийся в поле **zip** переменной **addr_info**.

Аналогично символьный массив **addr_info.name** можно использовать при вызове функции **gets()**.

```
gets(addr_info.name);
```

В этом операторе функции **gets()** передается указатель типа **char *** на начало массива **name**.

Поскольку переменная **name** представляет собой символьный массив, доступ к отдельным символам строки **addr_info.name** можно получить с помощью оператора индексирования. Например, в следующем фрагменте программы содержимое строки **addr_info.name** посимвольно выводится на экран.

```
register int t;

for(t=0; addr_info.name[t]; ++t)
  putchar(addr_info.name[t]);
```

Присваивание структур

Информацию, содержащуюся в одной структуре, можно присваивать другой структуре того же типа, используя обычный оператор присваивания. Иначе говоря, нет никакой необходимости присваивать каждый член отдельно. Рассмотрим программу, иллюстрирующую применение оператора присваивания к структурам.

```
#include <stdio.h>

int main(void)
{
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

  y = x;  /* Присваивание одной структуры другой */

  printf("%d", y.a);

  return 0;
}
```

После выполнения оператора присваивания член **y.a** будет содержать число 10.

Массивы структур

Чаще всего структуры используются как элементы массивов. Для того чтобы объявить массив структур, необходимо сначала определить структуру и объявить массив переменных этого типа. Например, чтобы объявить массив структур, состоящий из 100 элементов типа **addr**, необходимо выполнить оператор

```
struct addr addr_info[100];
```

Он создает набор, состоящий из 100 переменных, представляющих собой структуры типа **addr**.

Для того чтобы получить доступ к конкретной структуре, необходимо указать ее индекс. Например, чтобы вывести на экран почтовый индекс, хранящийся в третьей структуре, следует написать следующий оператор:

```
printf("%d", addr_info[2].zip);
```

Как во всех массивах, нумерация элементов массива структур начинается с нуля.

Передача структур функциям

В этом разделе рассматриваются вопросы, связанные с передачей структур и их членов функциям.

Передача членов структур

Если в функцию передается член структуры, на самом деле передается лишь копия его значения. Следовательно, в этом отношении член структуры ничем не отличается от обычной переменной (разумеется, если он сам не является составным элементом, например массивом). Рассмотрим следующую структуру.

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

Вот как ее члены передаются функциям.

```
func(mike.x);     /* Передается символ x */
func2(mike.y);    /* Передается целое число y */
func3(mike.z);    /* Передается число с плавающей точкой z */
func4(mike.s);    /* Передается адрес строки s */
func(mike.s[2]); /* Передается символ s[2] */
```

Если необходимо передать *адрес* отдельного члена структуры, следует указать оператор **&** перед именем структуры. Например, чтобы передать адреса членов структуры **mike**, нужно выполнить следующие операторы.

```
func(&mike.x);     /* Передается адрес символа x */
func2(&mike.y);    /* Передается адрес целого числа y */
func3(&mike.z);    /* Передается адрес числа с плавающей точкой z */
func4(mike.s);     /* Передается адрес строки  s */
func(&mike.s[2]); /* Передается адрес символа s[2] */
```

Обратите внимание на то, что оператор **&** стоит перед именем структуры, а не перед именами ее отдельных членов. Кроме того, имя строки **s** и так является адресом, поэтому указывать перед ней символ **&** не следует.

Передача целых структур

Если структура является аргументом функции, она передается по значению. Естественно, это значит, что все изменения структуры, происходящие внутри функции, никак не отразятся на структуре, являющейся ее фактическим аргументом.

Учтите, что в этом случае тип аргумента должен совпадать с типом параметра. Например, в следующей программе аргумент **arg** и параметр **parm** имеют одинаковый тип.

```
#include <stdio.h>

/* Определяем тип структуры. */
struct struct_type {
  int a, b;
  char ch;
} ;
```

```
void f1(struct struct_type parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg);

  return 0;
}

void f1(struct struct_type parm)
{
  printf("%d", parm.a);
}
```

Как видим, если параметры в функции объявлены как структуры, объявление их типа должно быть глобальным, чтобы все части программы могли использовать их по назначению. Например, если бы структура **struct_type** была объявлена внутри функции **main()**, она была бы невидимой в функции **f1()**.

Как уже указывалось, при передаче структур тип аргументов должен совпадать с типом параметров. Они должны быть не просто похожи, а идентичны. Например, следующий вариант предыдущей программы является неправильным и не будет скомпилирован, поскольку тип аргумента, указанный при вызове функции **f1()**, отличается от типа ее параметра.

```
/* Эта программа неверна и не будет скомпилирована. */
#include <stdio.h>

/* Определяем тип структуры. */
struct struct_type {
  int a, b;
  char ch;
} ;

/* Определяем тип, похожий на struct_type,
   но имеющий другое имя. */
struct struct_type2 {
  int a, b;
  char ch;
} ;

void f1(struct struct_type2 parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg); /* Несовпадение типов */
```

```
  return 0;
}

void f1(struct struct_type2 parm)
{
  printf("%d", parm.a);
}
```

Указатели на структуры

В языке C/C++ на структуры можно ссылаться точно так же, как и на любой другой тип данных. Однако указатели на структуры имеют несколько особенностей.

Объявление указателей на структуры

Указатели на структуры объявляются с помощью символа *****, стоящего перед именем экземпляра структуры. Например, указатель **addr_pointer** на структуру **addr** объявляется так:

```
struct addr *addr_pointer;
```

Учтите, что в языке C++ перед этим объявлением не обязательно указывать ключевое слово **struct**.

Использование указателей на структуры

Указатели на структуры используются в двух ситуациях: для передачи структуры в функцию по ссылке и для создания структур данных, основанных на динамическом распределении памяти (например, связанных списков). В этой главе мы рассмотрим лишь первое из этих применений.

Передача структур в качестве аргументов функции имеет один существенный недостаток: в стек функции приходится копировать целую структуру. (Напомним, что аргументы, передаваемые по значению, копируются в стек.) Если структура невелика, дополнительные затраты памяти будут относительно небольшими. Однако, если структура состоит из большого количества членов или ее членами являются большие массивы, затраты ресурсов могут оказаться чрезмерными. Этого можно избежать, если передавать функции не сами структуры, а лишь указатели на них.

Если функции передается указатель на структуру, в стек заталкивается только ее адрес. В результате вызовы функции выполняются намного быстрее. Второе преимущество, которое проявляется в некоторых случаях, заключается в том, что, получив адрес, функция может модифицировать структуру, являющуюся ее фактическим параметром.

Для того чтобы определить адрес структуры, достаточно поместить перед ее именем оператор **&**. Рассмотрим следующий фрагмент программы.

```
struct bal {
  float balance;
  char name[80];
} person;

struct bal *p;  /* Создаем указатель на структуру */
```

Теперь оператор

```
p = &person;
```

присваивает указателю **p** адрес структуры **person**.

Для того чтобы обратиться к элементу структуры через указатель на нее, нужно применить оператор "->". Например, вот как выглядит ссылка на поле **balance**:

```
p->balance
```

Оператор "->" обычно называют *оператором "стрелка"* (или *оператором ссылки на член структуры. — При. ред.*). Он состоит из символов "-" и ">". Если доступ к члену структуры осуществляется не непосредственно, а с помощью указателя, вместо оператора "." применяется оператор "->".

Проиллюстрируем применение указателя на структуру следующей программой, которая выводит на экран часы, минуты и секунды, используя системный таймер.

```
/* Отображение системного таймера. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
    display(&systime);
  }

  return 0;
}

void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}
```

```
void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
  long int t;

  /* change this as needed */
  for(t=1; t<DELAY; ++t) ;
}
```

Для настройки этой программы можно изменить определение константы **DELAY**.

Как видим, в программе определена глобальная структура **my_time**, но не объявлена переменная этого типа. В функции **main()** объявлена структура **systime**, инициализированная значением **00:00:30**. Это значит, что структура **systime** доступна только внутри функции **main()**.

Функция **update()**, изменяющая время, и функция **display()**, выводящая его на экран, получают адрес структуры **systime**. Аргумент обеих функций является указателем на структуру **my_time**.

В функциях **update()** и **display()** доступ к каждому члену структуры **systime** осуществляется через указатель. Поскольку функция **update()** получает указатель на структуру **systime**, она может изменять ее значение. Например, чтобы сбросить таймер на 0 по достижении момента времени 24:00:00, в функции **update()** предусмотрен оператор

```
if(t->hours==24) t->hours = 0;
```

Этот оператор вынуждает компилятор извлечь адрес, хранящийся в указателе **t**, ссылающемся на структуру **systime** в функции **main()**, и присвоить члену **hours** значение 0.

Помните, что оператор "**.**" используется для непосредственного обращения к членам структуры, а оператор "**->**" — для доступа к членам структуры через указатель на нее.

Массивы и структуры внутри структур

Членами структуры могут быть как простые, так и составные переменные. *Простыми* членами структур называются переменные, имеющие встроенный тип, например, целые числа или символы. С одной из разновидностей агрегированных элементов мы уже встречались, используя массив символов в структуре **addr**. Другим агрегированным типом данных могут быть одномерные и многомерные массивы, а также структуры.

Член структуры, представляющий собой массив, обрабатывается, как обычно. Рассмотрим следующую структуру.

```
struct x {
  int a[10][10]; /* Двухмерный массив целых чисел */
  float b;
} y;
```

Для того чтобы обратиться к элементу, индексы которого равны 3 и 7 соответственно, следует выполнить оператор

```
y.a[3][7]
```

Если членом структуры является другая структура, она называется *вложенной*. Например, структура **address** вложена в структуру **emp**:

```
struct emp {
  struct addr address; /* Вложенная структура */
  float wage;
} worker;
```

Здесь структура **emp** имеет два члена. Первым из них является структура типа **addr**, содержащая адрес сотрудника. Другим членом является переменная **wage**, в которой записан оклад сотрудника. Следующий фрагмент программы присваивает элементу **zip** структуры **address** потовый индекс 93456.

```
worker.address.zip = 93456;
```

Как видим, члены каждой структуры указываются слева направо в соответствии с глубиной вложения (от внешней — к внутренней). В языке C допускается до 15 уровней вложения. В языке C++ это количество увеличено до 256.

Битовые поля

В отличие от многих языков программирования, язык C/C++ обладает встроенной поддержкой *битовых полей* (bit-fields), предоставляющих доступ к отдельным битам. Битовые поля могут оказаться полезными во многих ситуациях.

- Если память ограничена, в одном байте можно хранить несколько логических переменных, принимающих значение **true** и **false**.
- Когда в одном байте нужно хранить информацию о состоянии некоторых устройств, закодированную несколькими битами.
- Если шифровальным процедурам требуется доступ к отдельным битам.

Хотя для решения этих задач можно использовать побитовые операторы, битовые поля позволяют создавать более простые и эффективные программы.

Для доступа к отдельным битам в языке C/C++ используется метод, основанный на структурах. Битовое поле представляет собой особую разновидность члена структуры, размер которого можно указывать в битах. Определение битового поля имеет следующий вид.

```
struct имя_типа_структуры
{
 тип имя1:длина;
 тип имя2:длина;
  .
  .
  .
 тип имя:длина;
} список_переменных;
```

Здесь спецификатор *тип* означает тип битового поля, а параметр *длина* — размер этого поля, выраженный в битах. Битовые поля, длина которых равна 1, должны быть объявлены с помощью спецификатора **unsigned**, поскольку отдельный бит не может иметь знака.

Битовые поля часто используют для анализа информации, поступающей от какого-либо устройства. Например, байт состояния последовательного порта организован следующим образом.

Бит	Смысл установленного бита
0	Изменение в линии сигнала разрешения на передачу (change in clear-to-send line).
1	Изменение состояния готовности устройства сопряжения (change in data-set-ready).
2	Обнаружен конец записи (trailing edge detected).
3	Изменение в приемной линии (change in receive line).
4	Разрешение на передачу (clear-to-send).
5	Готовность к приему (data-set-ready).
6	Телефонный звонок (telephone ringing).
7	Сигнал принят (received signal).

Информацию, записанную в байте состояния, можно представить в виде следующего битового поля.

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```

Для того чтобы позволить программе определять, можно ли посылать или получать данные, следует использовать такие операторы.

```
status = get_port_status();
if(status.cts) printf("Можно посылать");
if(status.dsr) printf("Можно принимать");
```

Чтобы присвоить битовому полю некое значение, используется тот же оператор, что и для обычного члена структуры. Например, в следующем фрагменте программы с помощью операторов доступа к члену структуры и оператора присваивания обнуляется поле **ring**.

```
status.ring = 0;
```

Как видим, для доступа к битовому полю используется оператор "**.**". Однако для доступа к битовому полю с помощью указателя на структуру необходимо применять оператор "**->**".

Имена битовым полям давать не обязательно. Это позволит обращаться только к нужным полям, игнорируя ненужные. Например, если вас интересуют только биты **cts** и **dsr**, структуру **status_type** можно объявить так.

```
struct status_type {
  unsigned :    4;
  unsigned cts: 1;
  unsigned dsr: 1;
} status;
```

Запомните: если биты, расположенные после бита **dsr**, не используются, указывать их не обязательно.

В одной и той же структуре обычные члены структуры используются наряду с битовыми полями. Например, в структуре

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:    1; /* уволен или работает */
```

```
  unsigned hourly:     1; /* почасовая оплата или оклад */
  unsigned deductions: 3; /* налоговые вычеты */
};
```

данные о сотруднике записаны в один байт, в котором хранится статус сотрудника, вид его зарплаты и размер налогов. Если бы эта структура не использовала битовые поля, ее размер был бы равен 3 байт.

Битовые поля накладывают некоторые ограничения. Например, нельзя получить адрес битового поля. Кроме того, битовые поля нельзя организовывать в массивы. Битовые поля нельзя объявлять статическими. Невозможно предугадать, будут ли биты считываться слева направо или справа налево при переносе на компьютер другого типа. Все это делает программы, использующие битовые поля, машинозависимыми. Помимо упомянутых, в разных операционных системах на битовые поля могут накладываться дополнительные ограничения.

Объединения

Объединение — это область памяти, которая используется для хранения переменных разных типов. Объединение позволяет интерпретировать один и тот же набор битов по-разному. Объявление объединения напоминает объявление структуры. Вот его общий вид.

```
union имя_типа_объединения
{
  тип имя_члена;
  тип имя_члена;
  тип имя_члена;
  .
  .
  .
} имена_экземпляров_объединения;
```

Например:

```
union u_type
{
  int i;
  char ch;
};
```

Это объявление не создает никаких переменных. Чтобы объявить экземпляр объединения, нужно либо указать его имя в конце объявления, либо применить отдельный оператор объявления. Рассмотрим пример программы на языке C, в котором объявляется экземпляр **cnvt** объединения **u_type**.

```
union u_type cnvt;
```

Для объявления экземпляра объединения в программе на языке C++ используется лишь имя его типа, ключевое слово **union** употреблять не обязательно. Например, вот как объявляется переменная **cnvt** в программе на языке C++.

```
u_type cnvt;
```

В языке C++ не запрещается указывать перед этим объявлением ключевое слово **union**, но это излишне, поскольку имя объединения полностью определяет его тип. В то же время, в языке C имя объединения считается дескриптором и поэтому должно сопровождаться ключевым словом **union**. (Ситуация аналогична объявлению структу-

ры.) Однако, поскольку программы, приведенные в этом разделе, должны быть правильными в обоих языках, мы придерживаемся стиля, принятого в языке С.

В переменной **cnvt** целочисленная переменная **i** и символьная переменная **ch** хранятся в одной и той же области памяти. Разумеется, переменная i занимает 2 байт (если размер типа **int** равен 2 байт), а переменная **ch** — только 1. На рис. 7.2 показано, каким образом обе переменные используют одну и ту же область памяти. В любом месте программы на переменную **cnvt** можно ссылаться, считая, что в ней хранится либо целое число, либо символ.

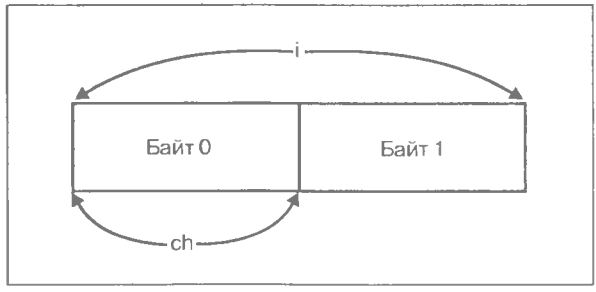


Рис. 7.2. Переменные **i** *и* **ch** *используют объединение* **cnvt** *(предполагается, что целое число занимает 2 байт)*

При объявлении экземпляра объединения компилятор автоматически выделяет память, достаточную для хранения наибольшего члена объединения. Например, если считать, что целые числа занимают 2 байт, размер переменной **cnvt** равен 2 байт, потому что в ней должна храниться целочисленная переменная **i**, хотя переменная **ch** занимает только один байт.

Для доступа к членам объединения используются те же синтаксические конструкции, что и для доступа к членам структуры: операторы "**.**" и "**->**". Для непосредственного обращения к членам объединения применяется оператор ".". Если доступ к экземпляру объединения осуществляется через указатель, используется оператор "→". Например, чтобы присвоить элементу **i** объединения **cnvt** число 10, следует выполнить следующий оператор.

```
cnvt.i = 10;
```

В следующем примере функции передается указатель на переменную **cnvt**.

```
void func1(union u_type *un)
{
  un->i = 10; /* присваиваем переменной cnvt значение 10,
                 используя функцию */
}
```

Объединения часто используются для специального преобразования типов, поскольку к хранящимся в объединении данным можно обращаться разными способами. Например, объединение можно использовать для манипуляции байтами, образующими значение типа **double**, например, изменять его точность или выполнять необычные округления.

Чтобы получить представление о полезности объединений при нестандартных преобразованиях типов, допустим, что нам нужно записать в файл значение переменной, имеющей тип **short int**. В языке C/C++ нет функции, предназначенной для записи в файл переменной типа **short int**. Разумеется, пользуясь функцией **fwrite()**, можно записать в файл значения любого типа, но для значений типа **short int** эта функция крайне неэффективна. Однако, используя объединение, можно легко написать функцию **putw()**, записывающую в файл двоичное представление значения типа

short int байт за байтом. (При этом считается, что целое число занимает два байта.) Чтобы понять, как это происходит, сначала создадим объединение, состоящее из переменной типа **short int** и двухбайтового символьного массива.

```
union pw {
  short int i;
  char ch[2];
};
```

Теперь используем объединение **pw** в функции **putw()**.

```
#include <stdio.h>

union pw {
  short int i;
  char ch[2];
};

int putw(short int num, FILE *fp);

int main(void)
{
  FILE *fp;

  fp = fopen("test.tmp", "wb+");

  putw(1000, fp);  /* Записываем число 1000 как целое */
  fclose(fp);

  return 0;
}

int putw(short int num, FILE *fp)
{
  union pw word;

  word.i = num;

  putc(word.ch[0], fp); /* Запись первой половины */
  return putc(word.ch[1], fp); /* Запись второй половины */
}
```

Несмотря на то что функция **putw()** вызывается с целочисленным аргументом, она без проблем вызывает стандартную функцию **putc()** для последовательной записи в файл байтов, из которых состоит целое число.

На заметку *В языке C++ предусмотрен особый вид объединений, называемых* ***безымянными*** *(anonimous). Они обсуждаются во второй части книги.*

Перечисления

Перечисление (enumeration) представляет собой набор именованных целочисленных констант, задающих все допустимые значения переменной данного типа. Перечисления часто встречаются в повседневной жизни. Например, названия монет, имеющих хождение в США, образуют перечисление

penny, nickel, dime, quarter, half-dollar, dollar

Перечисление определяется как структура и начинается с ключевого слова **enum** . Вот как выглядит перечисление:

enum *тип_перечисления* {*список_констант*} *список_переменных*

Тип_перечисления и *список_переменных* указывать не обязательно. (Однако хотя бы один из этих элементов объявления должен присутствовать.) Следующий фрагмент программы определяет перечисление с именем **coin**.

```
enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
```

Тип перечисления можно использовать для объявления переменных. В языке С оператор, приведенный ниже, объявляет переменную **money** типа **coin**.

```
enum coin money;
```

В языке C++ переменную **money** можно объявить с помощью более короткой записи:

```
coin money;
```

Кроме того, в языке C++ имя перечисления задает полный тип. В языке С оно является дескриптором, и поэтому для полного объявления необходимо использовать ключевое слово **enum**. (Ситуация аналогична объявлению структур и объединений.)

Итак, указанные ниже операторы являются абсолютно правильными.

```
money = dime;
if(money==quarter) printf("Это четвертак.\n");
```

Главное в перечислении то, что каждая константа обозначает целое число. Следовательно, их можно использовать вместо целочисленных переменных. Значение каждой константы на единицу превышает значение предыдущей. Первая константа равна 0. Таким образом, оператор

```
printf("%d %d", penny, dime);
```

выводит на экран числа **0 2**.

Значения констант можно задавать с помощью инициализатора. Для этого после имени константы следует поставить знак равенства и целое число. Константа, указанная после инициализатора, на единицу превышает значение, присвоенное предыдущей константе. Например, в следующем фрагменте программы константе **quater** присваивается значение 100.

```
enum coin { penny, nickel, dime, quarter=100,
            half_dollar, dollar};
```

Теперь значения констант распределены так:

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

Широко распространено заблуждение, что константы перечисления можно вводить и выводить как строки. Это не так. Например, фрагмент программы, приведенный ниже, работает неправильно.

```
/* Этот фрагмент работать не будет. */
money = dollar;
printf("%s", money);
```

Запомните, константа **dollar** — это просто имя целого числа, а не строка. По этой же причине следующий код также неверен.

```
/* Неверный код. */
strcpy(money, "dime");
```

Иначе говоря, строка, содержащая имя константы, не преобразовывается в строковую переменную автоматически.

Создать код, выполняющий ввод и вывод имен констант перечисления довольно сложно Это следует делать лишь тогда, когда вы не можете довольствоваться их целыми значениями. Например, в программе может возникнуть необходимость вывести на экран название монеты, а не ее код.

```
switch(money) {
  case penny: printf("penny");
    break;
  case nickel: printf("nickel");
    break;
  case dime: printf("dime");
    break;
  case quarter: printf("quarter");
    break;
  case half_dollar: printf("half_dollar");
    break;
  case dollar: printf("dollar");
}
```

Иногда для этого можно объявить массив строк и использовать константы перечисления в качестве индекса. Это позволит преобразовать значение константы в строку, содержащую ее имя.

```
char name[][12]={
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[money]);
```

Разумеется, этот способ можно применять, только если ни одна из констант перечисления не была инициализирована, поскольку массив строк индексируется с нуля, причем каждый следующий индекс увеличивается на 1.

Как видим, перечисления довольно трудно сделать наглядными, поэтому они особенно полезны там, где такие преобразования значений в имена не нужны. В частности, перечисления часто используются в компиляторе для создания таблицы символов. Кроме того, с их помощью можно верифицировать программы, поскольку перечисления гарантируют, что переменные могут принимать только предписанные значения.

Применение оператора sizeof для обеспечения машинонезависимости

Вы имели возможность убедиться, что структуры и объединения позволяют создавать переменные разных размеров, причем фактический размер переменных варьируется в зависимости от компьютера. Оператор **sizeof** позволяет вычислить размер лю-

бой переменной и сделать программу машинонезависимой. Этот оператор особенно полезен при работе со структурами и объединениями.

В ходе последующего изложения мы будем предполагать, что встроенные типы данных имеют следующие размеры, типичные для многих компиляторов языка C/C++.

Тип	*Размер в байтах*
char	1
int	4
double	8

Таким образом, фрагмент программы, приведенный ниже, выведет на экран числа 1, 4 и 8.

```
char ch;
int i;
double f;

printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

Размер структуры равен сумме размеров ее членов или *превышает его.* Рассмотрим пример.

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

Здесь число **sizeof(s_var)** равно, как минимум, 13 (8+4+1). Однако размер переменной **s_var** может быть больше, поскольку компилятор позволяет выравнивать структуру по границе слова или параграфа. (Параграф — это 16 байт.) Поскольку размер структуры может превышать сумму размеров ее членов, для определения ее фактического размера всегда следует применять оператор **sizeof**.

Поскольку оператор **sizeof** является статическим, вся информация, необходимая для вычисления размера любой переменной, становится известной уже на этапе компиляции. Это особенно удобно при работе с объединениями, поскольку их размер всегда равен размеру наибольшего члена. Рассмотрим пример.

```
union u {
  char ch;
  int i;
  double f;
} u_var;
```

Здесь число **sizeof(u_var)** равно 8. При выполнении программы уже не важно, что именно хранится в объединении **u_var** — главное, чтобы размер объединения совпадал с размером его наибольшего члена.

Оператор typedef

С помощью ключевого слова **typedef** можно определить новое имя типа данных. Новый тип при этом не *создается*, просто уже существующий тип получит новое имя. Это позволяет повысить машинонезависимость программ. Если в программе исполь-

зуется машинозависимый тип, достаточно его переименовать, и на новом компьютере для модификации программы придется изменить лишь одну строку с оператором **typedef**. Кроме того, с помощью оператора **typedef** можно давать типам осмысленные имена, что повышает наглядность программы. Общий вид оператора **typedef** таков:

typedef *тип новое_имя*

Здесь элемент *тип* обозначает любой допустимый тип, а элемент *новое_имя* — псевдоним этого типа. Новое имя является дополнительным. Оно не заменяет существующее имя типа.

Например, тип **float** можно переименовать следующим образом.

```
typedef float balance;
```

Этот оператор сообщает компилятору, что **balance** — это новое имя типа **float**. Теперь в программе вместо переменных типа **float** можно использовать переменные типа **balance**.

```
balance over_due;
```

Здесь переменная **over_due** является числом с плавающей точкой, но имеет тип **balance**, а не **float**. В свою очередь, используя оператор **typedef**, тип **balance** можно переименовать еще раз. Например, оператор

```
typedef balance overdraft;
```

сообщает компилятору, что **overdraft** — это синоним слова **balance**, которое является другим именем типа **float**.

Используя оператор **typedef**, можно повысить наглядность программы и ее машинонезависимость, но нельзя создать новый физический тип данных.

Полный справочник по C++

Глава 8

Ввод/вывод на консоль

В языке C++ предусмотрено две системы ввода-вывода. Первая система унаследована от языка C. Вторая, объектно-ориентированная система, присуща только языку C++. В этой и следующей главе мы рассмотрим средства ввода-вывода данных в языке C. (Систему ввода-вывода в языке C++ мы обсудим в части II.) Несмотря на то что в новых программах обычно используются объектно-ориентированные способы, старые средства ввода-вывода по-прежнему применяются широко. Кроме того, чтобы хорошо разбираться в новой системе ввода-вывода, определенной в языке C++, необходимо досконально понимать, как работает аналогичная система в языке C.

В языке C ввод и вывод данных осуществляются с помощью библиотечных функций, которые работают как с консолью, так и с файлами. С технической точки зрения консоль и файлы мало отличаются друг от друга, но концептуально они совершенно разные. В этой главе мы подробно рассмотрим функции, выполняющие ввод данных с клавиатуры и вывод данных на консоль. В следующей главе мы обсудим файловую систему языка C и проанализируем взаимосвязь обеих систем.

В этой главе описаны только консольные функции ввода-вывода, определенные стандартом языка C++, за одним исключением. В стандарте языка C++ не упомянута ни одна функция, управляющая дисплеем (например, функция управления курсором). Кроме того, в стандарте ничего не сказано о графических функциях. Это совершенно естественно — ведь упомянутые операции на разных компьютерах значительно отличаются друг от друга. Стандарт C++ совершенно игнорирует функции для вывода строк открытия диалоговых окон в системе Windows. Вместо этого в стандарте предполагается, что функции ввода-вывода выполняют только телетайпный вывод (TTY-output). Однако большинство компиляторов содержат в своих библиотеках функции для управления консолью, а также графические функции, предназначенные для использования в конкретной операционной системе. Разумеется, на языке C++ можно писать программы, работающие под управлением системы Windows. Просто следует иметь в виду, что специфические функции, ориентированные на конкретные операционные системы, стандартом языка C++ не предусмотрены.

Все функции ввода-вывода, определенные стандартом языка C, используют заголовочный файл **stdio.h**. В программах, написанных на языке C++, можно применять заголовочный файл **<cstdio>**.

Говоря о консольных функциях, мы имеем в виду функции для ввода данных с клавиатуры и для вывода данных на экран. Однако на самом деле эти функции работают со стандартными потоками, которые можно переназначать. Более того, стандартные потоки можно направлять на другие устройства (см. главу 9).

Важное замечание прикладного характера

В первой части книги мы рассматриваем систему ввода-вывода языка C, поскольку язык C является подмножеством языка C++. Как уже говорилось, в языке C++ определены свои собственные, объектно-ориентированные средства ввода-вывода. В большинстве приложений на языке C++ чаще всего используется именно эта система. Однако, чтобы хорошо понять основы ее функционирования, следует детально разобраться в системе ввода-вывода языка C. Для этого есть четыре причины.

- Когда-нибудь вам придется столкнуться со старыми программами, написанными на языке C.
- В обозримом будущем языки C и C++ будут сосуществовать. Многие программисты будут писать программы, используя гибрид C/C++. Кроме того, чаще всего на язык C++ переходят программисты, ранее работавшие на C. Следовательно, необходимо знать средства ввода-вывода данных, предусмотренные в обоих языках.

- Системы ввода-вывода данных в языке C и C++ основаны на одинаковых принципах.
- В некоторых ситуациях, например, при написании очень коротких программ, средства ввода-вывода языка C оказываются проще и нагляднее, чем их аналоги в языке C++.

Помимо всего прочего, существует неписаное правило, которое гласит: "Каждый программист на языке C++ должен уметь программировать на языке C". Отсутствие знаний о средствах ввода-вывода данных в языке C сужает ваш профессиональный кругозор.

Чтение и запись символов

Простейшими консольными функциями ввода и вывода являются функция **getchar()**, считывающая символ с клавиатуры, и функция **putchar()**, выводящая символ на экран. Функция **getchar()** ожидает нажатия клавиши и возвращает ее значение, которое автоматически выводится на экран. Функция **putchar()** выводит символ на экран в точку, определенную текущим положением курсора. Прототипы функций **getchar()** и **putchar()** выглядят так.

```
int getchar(void);
int putchar(inc c);
```

Как видим, функция **getchar()** возвращает целое число. В его младшем байте содержится код символа, соответствующего нажатой клавише. (Старший байт обычно содержит нулевое значение.) Это позволяет присвоить полученное целочисленное значение какой-нибудь символьной переменной. Если при вводе произошла ошибка, функция **getchar()** возвращает константу **EOF**.

Несмотря на то что функция **putchar()** по определению должна получать целочисленный аргумент, ей можно передавать только символьные значения. На экран выводится лишь младший байт аргумента. Функция **putchar()** возвращает либо символ, выведенный ею на экран, либо константу **EOF**, если произошла ошибка. (Константа **EOF** определена в заголовочном файле **stdio.h** и обычно равна -1).

Применение функций **getchar()** и **putchar()** иллюстрируется следующей программой. Она вводит символы с клавиатуры и выводит их в противоположном регистре, т.е. прописные буквы становятся строчными, и наоборот. Чтобы прекратить выполнение программы, достаточно ввести точку.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char ch;

  printf("Введите текст (для выхода введите точку).\n");
  do {
    ch = getchar();

    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

Проблемы, связанные с функцией getchar()

Функция **getchar()** может породить несколько проблем. Обычно эта функция помещает входные данные в буфер, пока не будет нажата клавиша <ENTER>. Такой способ называется *буферизованным вводом* (line-buffered input). Для того чтобы данные, которые вы ввели, действительно были переданы программе, следует нажать клавишу <ENTER>. Кроме того, при каждом вызове функция **getchar()** вводит символы по одному, последовательно размещая их в очереди. Если программа использует интерактивный диалог, такое торможение становится раздражающим фактором. Несмотря на то что стандарт языка C/C++ позволяет сделать функцию **getchar()** интерактивной, эта возможность используется редко. Итак, если вдруг обнаружится, что приведенная выше программа работает не так, как ожидалось, вы будете знать причину.

Альтернативы функции getchar()

Функция **getchar()**, реализованная вашим компилятором, может не соответствовать требованиям интерактивной среды. В этом случае можно использовать другие функции, позволяющие считывать символы с клавиатуры. В стандарте языка C++ не предусмотрено ни одной функции, гарантирующей интерактивный ввод, но на практике этот недостаток восполняется компиляторами.

Наиболее известными альтернативами являются функции **getch()** и **getche()**. Их прототипы выглядят так.

```
int getch(void);
int getche(void);
```

Большинство компиляторов размещают прототипы этих функций в заголовочном файле **conio.h**. В некоторых компиляторах перед именами этих функций ставится знак подчеркивания. Например, в компиляторе Microsoft Visual C++ эти функции называются **_getch()** и **_getche()** соответственно.

После нажатия клавиши функция **getch()** немедленно возвращает результат, введенный символ на экране не отображается. Функция **getche()** аналогична функции **getch()**, за одним исключением: введенный символ отображается на экране. В интерактивных программах вместо функции **getchar()** чаще всего используются функции **getch()** и **getche()**. Однако, если компилятор не поддерживает эти функции, либо функция **getchar()**, реализованная им, полностью соответствует требованиям интерактивной среды, следует применять именно ее.

Перепишем предыдущую программу, заменяя функцию **getchar()** функцией **getch()**.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

int main(void)
{
  char ch;

  printf("Введите текст (для выхода введите точку).\n");
  do {
    ch = getch();
```

```
    if(islower(ch)) ch = toupper(ch);
    else ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

В ходе выполнения этой программы при каждом нажатии клавиши соответствующий символ будет немедленно передан программе и отображен на экране. Ввод не буферизуется. Хотя в других программах, рассмотренных в этой книге, мы больше не будем использовать функции **getch()** и **getche()**, о них всегда следует помнить.

На заметку *Когда создавалась эта книга, функции **_getche()** и **_getch()**, предусмотренные компилятором Microsoft Visual C++, не были совместимы со стандартными функциями ввода-вывода, такими как **scanf()** или **gets()**. Вместо них приходится использовать специальные версии стандартных функций, например **cscanf()** и **cgets()**. Детали вы найдете в документации по компилятору Visual C++.*

Чтение и запись строк

Следующими по сложности и эффективности являются консольные функции **gets()** и **puts()**. Они позволяют считывать и записывать строки.

Функция **gets()** считывает строку символов, введенных с клавиатуры, и размещает их по адресу, указанному в аргументе. Символы на клавиатуре набираются до тех пор, пока не будет нажата клавиша <ENTER>. В конец строки ставится не символ перехода на новую строку, а нулевой байт. После этого функция **gets()** завершает свою работу. Функцию **gets()** в отличие от функции **getchar()** нельзя использовать для перехода на новую строку. Ошибки, допущенные при наборе строки, можно исправить с помощью клавиши <BACKSPACE>. Прототип функции **gets()** имеет следующий вид.

```
char *gets(char *строка)
```

Здесь параметр *строка* представляет собой массив, в который записываются символы, введенные пользователем. Следующая программа считывает строку в массив **str** и выводит на экран его длину.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];

  gets(str);
  printf("Длина массива равна %d", strlen(str));

  return 0;
}
```

При работе с функцией **gets()** следует проявлять осторожность, поскольку она не проверяет выход индекса массива за пределы допустимого диапазона. Следовательно, количество символов, введенных пользователем, может превысить длину массива. Хотя функция **gets()** прекрасно работает в простых программах, ее не следует приме-

нять в коммерческих приложениях. Ее альтернативой является функция **fgets()**, описанная в следующей главе. Эта функция предотвращает переполнение массива.

Функция **puts()** выводит на экран строку символов и переводит курсор на следующую строку экрана. Ее прототип выглядит так.

```
int puts(const char *строка)
```

Функция **puts()**, как и функция **printf()**, распознает эскейп-последовательности, например **'\n'**, предназначенную для перехода на новую строку. На вызов функции **puts()** тратится меньше ресурсов, чем на функцию **printf()**, поскольку она может выводить лишь строки, но не числа. Кроме того, она не форматирует вывод. Таким образом, функция **puts()** занимает меньше места и выполняется быстрее, чем функция **printf()**. По этой причине функцию **puts()** часто применяют для оптимизации кода. Если при выводе возникла ошибка, функция **puts()** возвращает константу **EOF**. В противном случае она возвращает неотрицательное значение. Однако при выводе данных на консоль программисты редко учитывают возможность ошибки, поэтому значение, возвращаемое функцией **puts()**, проверяется редко. Следующий фрагмент выводит на экран слово "Привет".

```
puts("Привет");
```

Сведения об основных консольных функциях приведены в табл. 8.1.

Таблица 8.1. Основные функции ввода-вывода

Функция	Операция
getchar()	Считывает символ с клавиатуры; ожидает перехода на новую строку
getche()	Считывает символ с клавиатуры и выводит его на экран; не ожидает перехода на новую строку; не определена в стандарте языка C/C++, но широко используется
getch()	Считывает символ с клавиатуры и не выводит его на экран; не ожидает перехода на новую строку; не определена в стандарте языка C/C++, но широко используется
putchar()	Выводит символ на экран
gets()	Считывает строку с клавиатуры
puts()	Выводит строку на экран

Следующая программа демонстрирует применение нескольких основных консольных функций ввода-вывода при работе с компьютерным словарем. Программа предлагает пользователю ввести слово, а затем проверяет, содержится ли это слово в базе данных. Если слово найдено, на экран выводится его значение. Обратите особое внимание на косвенную адресацию. Если вам не понятно, как она применяется, вспомните, что переменная **dic** является массивом указателей на строки.

```
/* Простой словарь. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/* Список слов и их значений */
char  *dic[][40] = {
  "автомобиль", "Транспортное средство с двигателем.",
  "атлас", "Собрание карт, изданное в виде книги.",
  "аэроплан", "Летающая машина."
  "телефон", "Средство связи.",
  "", ""  /* нулевой символ, завершающий список */
};
```

```
int main(void)
{
  char word[80], ch;
  char **p;

  do {
    puts("\nВведите слово: ");
    scanf("%s", word);

    p = (char **)dic;

    /* Поиск соответствия и вывод значения */
    do {
      if(!strcmp(*p, word)) {
        puts("Значение:");
        puts(*(p+1));
        break;
      }
      if(!strcmp(*p, word)) break;
      p = p + 2;  /* Перемещение вперед по списку */
    } while(*p);
    if(!*p) puts("Слова в словаре нет.");
    printf("Продолжить работу? (y/n): ");
    scanf(" %c%*c", &ch);
  } while(toupper(ch) != 'N');

  return 0;
}
```

Форматированный ввод-вывод на консоль

Функции **printf()** и **scanf()** выполняют форматированный ввод-вывод на консоль, иначе говоря, они могут считывать и записывать данные в заданном формате. Функция **printf()** выводит данные на консоль. Функция **scanf()**, наоборот, считывает данные с клавиатуры. Обе функции могут оперировать любыми встроенными типами данных, включая символы, строки и числа.

Функция printf()

Прототип функции **printf()** выглядит следующим образом.

```
int printf(const char *управляющая_строка, ...)
```

Функция **printf()** возвращает количество записанных ею символов, а в случае ошибки — отрицательное число.

Параметр *управляющая_строка* состоит из элементов двух видов. Во-первых, он содержит символы, которые выводятся на экран. Во-вторых, в него входят спецификаторы формата, начинающиеся со знака процента, за которым следует код формата. Количество аргументов должно совпадать с количеством спецификаторов формата, причем они попарно сравниваются слева направо. Например, вызов

```
printf("Я люблю %c%s", 'C', "++!");
```

выведет на экран строку

Функция **printf()** допускает широкий выбор спецификаторов формата, показанных в табл. 8.2.

Таблица 8.2. Спецификаторы формата функции `printf()`

Код	*Формат*
`%c`	Символ
`%d`	Десятичное целое число со знаком
`%i`	Десятичное целое число со знаком
`%e`	Научный формат (строчная буква e)
`%E`	Научный формат (прописная буква E)
`%f`	Десятичное число с плавающей точкой
`%g`	В зависимости от того, какой формат короче, применяется либо `%e`, либо `%f`
`%G`	В зависимости от того, какой формат короче, применяется либо `%E`, либо `%F`
`%o`	Восьмеричное число без знака
`%s`	Строка символов
`%u`	Десятичное целое число без знака
`%x`	Шестнадцатеричное число без знака (строчные буквы)
`%X`	Шестнадцатеричное число без знака (прописные буквы)
`%p`	Указатель
`%n`	Указатель на целочисленную переменную. Спецификатор вызывает присваивание этой целочисленной переменной количества символов, выведенных перед ним
`%%`	Знак `%`

Вывод символов

Для вывода отдельных символов используется спецификатор **%c**. В результате соответствующий аргумент без изменений будет выведен на экран.

Для вывода строк применяется спецификатор **%s**.

Вывод чисел

Для вывода десятичных целых чисел со знаком применяются спецификаторы **%d** или **%i**. Эти спецификаторы эквивалентны. Одновременная поддержка обоих спецификаторов обусловлена историческими причинами.

Для вывода целого числа без знака следует применять спецификатор **%u**.

Спецификатор формата **%f** позволяет выводить на экран числа с плавающей точкой.

Спецификаторы **%e** и **%E** сообщают функции **printf()**, что на экран выводится аргумент типа **double** в научном формате. Числа, представленные в научном формате, выглядят так:

```
x.ddddd E+/-yy
```

Если буква E должна быть выведена как прописная, следует использовать спецификатор **%E**, а если как строчная — **%e**.

Функция **printf()** может сама выбирать представление числа с помощью спецификатора **%f** или **%e**, если вместо них указать спецификаторы **%g** или **%G**. В этом случае функция сама определит, какой вид числа короче. Спецификатор **%G** позволяет вывести букву **E** как прописную, а **%g** — как строчную. Следующая программа демонстрирует эффект применения спецификатора **%g**.

```
#include <stdio.h>

int main(void)
{
  double f;

  for(f=1.0; f<1.0e+10; f=f*10)
    printf("%g ", f);

  return 0;
}
```

В результате на экране появятся такие числа.

```
1 10 100 1000 10000 100000 1e+006 1e+007 1e+008 1e+009
```

Целые числа без знака можно выводить в восьмеричном и шестнадцатеричном формате, используя спецификаторы **%o** и **%x**. Поскольку в шестнадцатеричной системе счисления для представления чисел от 10 до 15 используются буквы от A до F, их можно выводить как прописными, так и строчными. Если буквы должны быть прописными, следует применять спецификатор **%X**, а если строчными — **%x**, как показано ниже.

```
#include <stdio.h>

int main(void)
{
  unsigned num;

  for(num=0; num<255; num++) {
    printf("%o ", num);
    printf("%x ", num);
    printf("%X\n", num);
  }

  return 0;
}
```

Вывод адресов

Если на экран необходимо вывести адрес, следует применять спецификатор **%p**. Этот спецификатор формата заставляет функцию **printf()** выводить на экран адрес, формат которого совместим с типом адресации, принятой в компьютере. Следующая программа выводит на экран адрес переменной **sample**.

```
#include <stdio.h>

int sample;

int main(void)
{
  printf("%p", &sample);

  return 0;
}
```

Спецификатор %n

Спецификатор формата **%n** отличается ото всех остальных. Он заставляет функцию **printf()** записывать в соответствующую переменную количество символов, уже вы-

веденных на экран. Спецификатору `%n` должен соответствовать целочисленный указатель. После завершения функции **`printf()`** этот указатель будет ссылаться на переменную, в которой содержится количество символов, выведенных до спецификатора `%n`. Этот необычный спецификатор иллюстрируется следующим примером.

```
#include <stdio.h>

int main(void)
{
  int count;

  printf("Это%n проверка\n", &count);
  printf("%d", count);

  return 0;
}
```

Эта программа выведет на экран строку "Это проверка" и число 3. Спецификатор `%n` обычно используется для динамического форматирования.

Модификаторы формата

Многие спецификаторы формата имеют свои модификаторы, которые немного изменяют их смысл. Например, с их помощью можно изменять минимальную ширину поля, количество цифр после десятичной точки, а также выполнять выравнивание по левому краю. Модификатор формата указывается между символом процента и кодом формата. Рассмотрим их подробнее.

Модификатор минимальной ширины поля

Целое число, размещенное между символом процента и кодом формата, задает *минимальную ширину поля*. Если строка вывода короче, чем нужно, она дополняется пробелами, если длиннее, строка все равно выводится полностью. Строку можно дополнять не только пробелами, но и нулями. Для этого достаточно поставить 0 перед модификатором ширины поля. Например, спецификатор **`%05d`** дополнит число, количество цифр которого меньше пяти, ведущими нулями, так что в результате оно будет состоять из пяти цифр. Этот спецификатор иллюстрируется следующей программой.

```
#include <stdio.h>

int main(void)
{
  double item;

  item = 10.12304;

  printf("%f\n", item);
  printf("%10f\n", item);
  printf("%012f\n", item);

  return 0;
}
```

Эта программа выводит на экран следующие числа.

```
10.123040
 10.123040
00010.123040
```

Модификатор минимальной ширины поля чаще всего используется для форматирования таблиц. Программа, приведенная ниже, создает таблицу квадратов и кубов чисел от 1 до 19.

```
#include <stdio.h>

int main(void)
{
  int i;

  /* выводит таблицу квадратов и кубов от 1 до 19 */
  for(i=1; i<20; i++)
    printf("%8d %8d %8d\n", i, i*i, i*i*i);

  return 0;
}
```

В результате на экран будет выведена следующая таблица.

```
 1       1       1
 2       4       8
 3       9      27
 4      16      64
 5      25     125
 6      36     216
 7      49     343
 8      64     512
 9      81     729
10     100    1000
11     121    1331
12     144    1728
13     169    2197
14     196    2744
15     225    3375
16     256    4096
17     289    4913
18     324    5832
19     361    6859
```

Модификатор точности

Модификатор точности указывается после модификатора ширины поля (если он есть). Этот модификатор состоит из точки, за которой следует целое число. Точный смысл модификатора зависит от типа данных, к которым он применяется.

Если модификатор точности применяется к числам с плавающей точкой с форматами **%f**, **%e** или **%E**, он означает количество десятичных цифр после точки. Например, спецификатор формата **%10.4f** означает, что на экран будет выведено число, состоящее из десяти цифр, четыре из которых расположены после точки.

Если модификатор применяется к спецификаторам формата **%g** или **%G**, он задает количество значащих цифр.

Если модификатор используется для вывода строк, он задает максимальную длину поля. Например, спецификатор **%5.7s** означает, что на экран будет выведена строка, состоящая как минимум из пяти символов, длина которой не превышает семи. Если строка окажется длиннее, последние символы будут отброшены.

Если модификатор точности применяется к целым типам, он задает минимальное количество цифр, из которых должно состоять число. Если число состоит из меньшего количества цифр, оно дополняется ведущими нулями.

Рассмотрим демонстрационную программу.

```
#include <stdio.h>

int main(void)
{
  printf("%.4f\n", 123.1234567);
  printf("%3.8d\n", 1000);
  printf("%10.15s\n", "Это простая проверка.");

  return 0;
}
```

Эта программа выводит на экран следующие результаты.

```
123.1235
00001000
Это простая про
```

Выравнивание вывода

По умолчанию вывод выравнивается по правому краю. Иначе говоря, если ширина поля больше, чем выводимые данные, результаты "прижимаются" к правому краю. Вывод можно выровнять по левому краю, поставив перед символом **%** знак "минус". Например, спецификатор **%-10.2f** выравнивает число с двумя знаками после точки по левому краю поля, состоящего из 10 позиций.

Рассмотрим пример.

```
#include <stdio.h>

int main(void)
{
  printf("Выравнивание по правому краю:%8d\n", 100);
  printf("Выравнивание по левому краю:%-8d\n", 100);

  return 0;
}
```

Обработка данных других типов

Для вывода значений переменных типа **short int** и **long int** функция **printf()** использует два модификатора, которые можно применять к спецификаторам **d**, **i**, **o**, **u** и **x**. Модификатор **l** (буква "эль") позволяет выводить значения типа **long int**. Например, спецификатор **%ld** означает, что на экран будет выведено значение типа **long int**. Для вывода значения типа **short int** используется модификатор **h**. Например, спецификатор **%hu** означает, что на экран будет выведено значение типа **short unsigned int**.

Модификаторы **l** и **h** можно также применять к спецификатору **n**. В этом случае соответствующий указатель должен иметь тип **long int*** или **short int***.

Если компилятор полностью поддерживает стандарт языка C++, модификатор **l** можно применять к спецификатору **c**. В этом случае он задает расширенный символ. Аналогично, если модификатор **l** стоит перед спецификатором **s**, на экран выводится строка расширенных символов.

Модификатор **L** можно использовать как префикс перед спецификаторами **e**, **f** и **g**. Он означает, что на экран выводится значение типа **long double**.

Модификаторы * и

Функция **printf()** имеет еще два модификатора: * и #.

Если перед спецификаторами **g**, **G**, **f**, **E** или **e** стоит модификатор #, это означает, что число будет содержать десятичную точку, даже если оно не имеет дробной части. Если этот модификатор стоит перед спецификаторами **x** или **X** , шестнадцатеричные числа выводятся с префиксом **0x**. Если же символ # указан перед спецификатором **o**, число будет дополнено ведущими нулями. К другим спецификаторам модификатор # применять нельзя.

Ширину поля и точность представления числа можно задавать не только константами, но и переменными. Для этого вместо точных значений в спецификаторе следует указать символ *. При сканировании строки вывода функция **printf()** поочередно сопоставляет модификатор * с каждым аргументом. Например, на рис. 8.1 показан вывод числа с четырьмя цифрами после точки в поле шириной 10 символов. В результате на экран будет выведено число **123.3**.

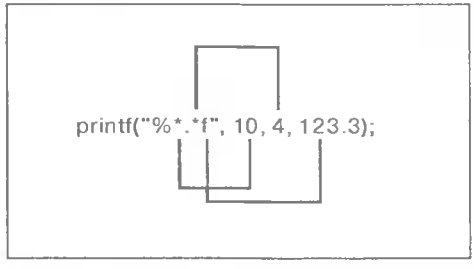


*Рис. 8.1. Сопоставление модификатора * с его значениями*

Следующая программа иллюстрирует применение модификаторов # и *.

```
#include <stdio.h>

int main(void)
{
  printf("%x %#x\n", 10, 10);
  printf("%*.*f", 10, 4, 1234.34);

  return 0;
}
```

Функция scanf()

Функция **scanf()** представляет собой процедуру ввода. Она может считывать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат. Данная функция выглядит полной противоположностью функции **printf()**. Прототип функции **scanf()** имеет следующий вид.

```
int scanf(const char *управляющая_строка, ...)
```

Функция **scanf()** возвращает количество переменных, которым она успешно присвоила свои значения. Если при чтении произошла ошибка, функция **scanf()** возвращает константу **EOF**. Параметр *управляющая_строка* определяет порядок считывания значений и присваивания их переменным, указанным в списке аргументов.

Управляющая строка состоит из символов, разделенных на три категории.

- Спецификаторы формата.
- Разделители.
- Символы, не являющиеся разделителями

Рассмотрим каждую из этих категорий.

Спецификаторы формата

Перед спецификаторами формата стоит символ `%`. Они сообщают функции `scanf()` тип данных, подлежащих вводу. Эти коды перечислены в табл. 8.3. Спецификаторы формата сопоставляются с аргументами слева направо. Рассмотрим несколько примеров.

Таблица 8.3. Спецификаторы формата функции `scanf()`

Код	*Формат*
`%c`	Символ
`%d`	Десятичное целое число со знаком
`%i`	Десятичное целое число со знаком, восьмеричное или шестнадцатеричное число
`%e`	Десятичное число с плавающей точкой
`%f`	Десятичное число с плавающей точкой
`%g`	Десятичное число с плавающей точкой
`%o`	Восьмеричное число
`%s`	Строка символов
`%u`	Десятичное целое число без знака
`%x`	Шестнадцатеричное число без знака (строчные буквы)
`%p`	Указатель
`%n`	Указатель на целочисленную переменную. Спецификатор вызывает присваивание этой целочисленной переменной количества символов, введенных перед ней
`%[]`	Набор сканируемых символов
`%%`	Знак `%`

Ввод чисел

Для ввода целого числа используются спецификаторы `%d` или `%i`. Для ввода числа с плавающей точкой, представленной в стандартном или научном формате, применяются спецификаторы `%e`, `%f` или `%g`.

Используя спецификаторы `%o` или `%x`, можно вводить целые числа, представленные в восьмеричном или шестнадцатеричном формате соответственно. Спецификатор `%x` имеет два варианта, предназначенных для ввода строчных и прописных шестнадцатеричных цифр от A до F. Однако при вводе шестнадцатеричных чисел это не имеет значения. Рассмотрим программу, выполняющую ввод восьмеричного и шестнадцатеричного числа.

```
#include <stdio.h>

int main(void)
{
  int i, j;

  scanf("%o%x", &i, &j);
  printf("%o %x", i, j);
```

```
  return 0;
}
```

Функция **scanf()** прекращает вводить числа, обнаружив первый нечисловой символ.

Ввод целых чисел без знака

Для ввода целых чисел без знака применяется модификатор **%u**. Например, фрагмент

```
unsigned num;
scanf("%u", &num);
```

вводит целое число без знака и присваивает его переменной **num**.

Ввод отдельных символов

Как указывалось ранее, отдельные символы можно вводить с помощью функции **getchar()** или производных от нее функций. Функцию **scanf()** также можно применять для этой цели, используя спецификатор **%c**. Однако, как и функция **getchar()**, функция **scanf()** использует буферизованный ввод, поэтому в интерактивных программах ее применять не следует.

Несмотря на то что пробелы, знаки табуляции и символы перехода на новую строку используются как разделители при чтении данных любых типов, при вводе отдельных символов они считываются наравне со всеми. Например, если поток ввода содержит строку **"x y"**, то фрагмент кода

```
scanf("%c%c%c", &a, &b, &c);
```

присвоит символ **x** переменной **a**, пробел — переменной **b** и символ **y** — переменной **c**.

Ввод строк

Функцию **scanf()** можно применять для ввода строк из входного потока. Для этого используется спецификатор **%s**. Он заставляет функцию **scanf()** считывать символы, пока не обнаружится разделитель. Символы, считанные из входного потока, записываются в массив, на который ссылается соответствующий аргумент, а в конец этого массива записывается нулевой байт. Функция **scanf()** считает разделителем пробел, символ перехода на новую строку, символ табуляции, символ вертикальной табуляции, а также символ прогона бумаги. Следовательно, функцию **scanf()** нельзя просто применить для ввода строки "Это проверка", поскольку ввод прекратится на первом же пробеле. Чтобы увидеть эффект, произведенный спецификатором **%s**, попробуйте применить следующую программу к строке "Всем привет".

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Введите строку: ");
  scanf("%s", str);
  printf("Вот ваша строка: %s", str);

  return 0;
}
```

Программа вернет лишь слово "Всем", а остальную часть строки проигнорирует.

Ввод адреса

Для ввода адреса применяется спецификатор **%p**. Он заставляет функцию **scanf()** считывать из входного потока адрес, формат которого определяется архитектурой центрального процессора. Например, следующая программа вводит адрес ячейки, а затем выводит на экран ее содержимое.

```
#include <stdio.h>

int main(void)
{
  char *p;

  printf("Введите адрес: ");
  scanf("%p", &p);
  printf("По этому адресу хранится значение%p is %c\n", p, *p);

  return 0;
}
```

Спецификатор %n

Спецификатор **%n** вынуждает функцию **scanf()** присвоить количество ранее введенных символов переменной, на которую ссылается соответствующий аргумент.

Использование набора сканируемых символов

Функция **scanf()** поддерживает спецификатор универсального формата, называемый *набором сканируемых символов* (scanset). При обработке этого спецификатора функция будет вводить только те символы, которые входят в заданный набор. Символы записываются в массив, на который ссылается соответствующий аргумент. Для того чтобы определить набор сканируемых символов, достаточно перечислить их в квадратных скобках. Перед открывающей квадратной скобкой указывается символ процента. Например, следующий набор сканируемых символов означает, что функция **scanf()** должна вводить лишь символы **X**, **Y** и **Z**.

```
%[XYZ]
```

При использовании набора сканируемых символов функция **scanf()** продолжает считывать символы, помещая их в соответствующий массив, пока не обнаружится символ, не входящий в заданный набор. По возвращении из функции **scanf()** массив будет содержать строку, состоящую из считанных элементов и завершающуюся нулевым байтом. Все вышесказанное иллюстрируется следующей программой.

```
#include <stdio.h>

int main(void)
{
  int i;
  char str[80], str2[80];

  scanf("%d%[abcdefg]%s", &i, str, str2);
  printf("%d %s %s", i, str, str2);

  return 0;
}
```

Введите строку **123abcdtye** и нажмите клавишу <ENTER>. Программа выведет на экран следующие строки: **123 abcd tye**. Поскольку символ **t** не входит в набор сканируемых символов, обнаружив его, функция **scanf()** останавливает считывание данных из входного потока в переменную **str**. Оставшиеся символы считываются в переменную **str2**.

Если первым символом набора является знак **^**, набор сканируемых символов трактуется наоборот. Знак **^** сообщает функции **scanf()**, что она должна вводить только символы, *не* определенные в наборе.

В большинстве реализаций можно задавать диапазон символов, используя дефис. Например, следующий спецификатор ограничивает ввод символов диапазоном от **A** до **Z**.

```
%[A-Z]
```

Следует помнить, что набор сканируемых символов чувствителен к регистру. Для того чтобы ввести как прописные, так и строчные буквы, их следует указать отдельно.

Пропуск нежелательных разделителей

Если в управляющей строке задан разделитель, функция **scanf()** пропускает один или несколько разделителей во входном потоке. Разделителем может быть пробел, символ табуляции, символ вертикальной табуляции, символ прогона бумаги, а также символ перехода на новую строку. По существу, один разделитель в управляющей строке заставляет функцию **scanf()** считывать, но не записывать в память любое количество разделителей, стоящих перед первым символом, не являющимся разделителем.

Символы, не являющиеся разделителями

Символы, не являющиеся разделителями, вынуждают функцию **scanf()** считывать и пропускать заданные символы. Например, управляющая строка **"%d,%d"** заставляет функцию **scanf()** считать целое число, прочитать и отбросить запятую, а затем считать следующее целое число. Если указанный символ не найден, функция **scanf()** прекращает работу. Если нужно считать и отбросить знак процента, в управляющей строке используется обозначение **%%**.

Функции scanf() следует передавать адреса

Аргументы функции **scanf()** должны передаваться по ссылке. Таким образом, ее аргументами должны быть указатели. Напомним, что только этот способ передачи аргументов позволяет функции изменять их значения. Например, для того, чтобы ввести целое число и присвоить его переменной **count**, функцию **scanf()** следует вызвать следующим образом.

```
scanf("%d", &count);
```

Строки считываются в символьные массивы, имена которых сами являются их адресами. Следовательно, для считывания строки в массив **str** необходимо использовать вызов

```
scanf("%s", str);
```

В данном случае переменная **str** сама является указателем, и ставить перед ней оператор взятия адреса **&** не следует.

Модификаторы формата

Как и функция **printf()**, функция **scanf()** допускает модификацию спецификаторов формата.

Спецификаторы формата могут содержать модификатор максимальной ширины поля. Он представляет собой целое число, указанное между знаком процента и спецификатором формата. Это число ограничивает количество символов, которое можно ввести. Например, чтобы в строку **str** нельзя было ввести больше 20 символов, следует выполнить следующий вызов.

```
scanf("%20s", str);
```

Если длина строки во входном потоке больше 20, следующий ввод начнется с того места, где остановился предыдущий. Например, при вводе строки

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

предыдущий оператор введет лишь двадцать символов, т.е. до буквы "**T**" включительно, и запишет их в массив **str**. Остальные символы, т.е. **UVWXYZ**, не будут введены. Если после этого будет выполнен новый вызов функции **scanf()**, например,

```
scanf("%s", str);,
```

то буквы **UVWXYZ** будут записаны в переменную **str**. Ввод символов в ограниченное поле прекращается, либо если достигнута максимальная ширина поля, либо если обнаружен разделитель. В этом случае функция **scanf()** переходит к следующему полю.

Для ввода значений типа **long int** перед спецификатором формата следует поставить модификатор **l**. Для ввода значений типа **short int** перед спецификатором формата следует указать модификатор **h**. Эти модификаторы можно ставить перед спецификаторами **d**, **i**, **o**, **u**, **x** и **n**.

По умолчанию спецификаторы **f**, **e** и **g** сообщают функции **scanf()**, что вводится число с плавающей точкой. Если перед этими спецификаторами стоит буква **l**, введенное значение будет присвоено переменной типа **double**. Префикс **L** означает ввод значения типа **long double**.

Подавление ввода

Можно заставить функцию **scanf()** считывать поле, но не присваивать его ни одной переменной. Для этого перед кодом соответствующего формата следует поставить символ *. Например, вызов

```
scanf("%d%*c%d", &x, &y);
```

означает, что в программу вводится пара чисел **10,10**. Запятая между числами должна считываться, но не присваиваться ни одной переменной. Подавление присваивания особенно полезно, когда программа обрабатывает лишь часть входной информации.

Полный справочник по

Глава 9

Файловый ввод/вывод

В этой главе описывается файловая система языка С. Как указывалось в главе 8, язык С++ поддерживает две системы ввода-вывода: одна из них унаследована от языка С, а другая является объектно-ориентированной. В этой главе мы рассмотрим лишь первую из этих систем. (Файловая система, характерная только для языка С++, будет описана в части II.) Несмотря на то что в большинстве современных программ для ввода-вывода используется в основном файловая система языка С++, программисты по-прежнему должны хорошо разбираться в средствах ввода-вывода, предусмотренных языком С. Причины этого мы уже указали в предыдущей главе.

Файловые системы языков С и С++

Некоторые программисты имеют неверное представление о связях между файловыми системами языка С и С++. Во-первых, язык С++ полностью поддерживает файловую систему языка С. Следовательно, при переводе программ с языка С на С++ процедуры ввода-вывода можно оставить без изменения. Во-вторых, в языке С++ определена собственная, объектно-ориентированная, система ввода-вывода, включающая функции и операторы. Эта система полностью дублирует все функциональные возможности средств ввода-вывода языка С, что делает их излишними. Программист может совершенно свободно применять обе файловые системы. И все же большинство программистов предпочитают использовать систему ввода-вывода языка С++. Причины такого выбора станут ясны после изучения части II.

Потоки и файлы

Прежде чем перейти к обсуждению файловой системы языка С, следует понять разницу между *потоками* и *файлами*. Система ввода-вывода языка С обеспечивает единообразный интерфейс, не зависящий от физических устройств. Иначе говоря, система ввода-вывода создает между программистом и устройством абстрактное средство связи. Эта *абстракция* называется *потоком*, а физическое устройство — *файлом*. Взаимодействие между потоками и файлами представляет для нас особый интерес.

На заметку *В системе ввода-вывода языка С++ понятия потока и файла также играют важную роль.*

Потоки

Файловая система языка С предназначена для широкого спектра средств ввода-вывода, включая терминалы, дисководы и принтеры. Несмотря на разнообразие физических устройств, файловая система преобразовывает каждое из них в некое логическое устройство, называемое *потоком*. Все потоки функционируют одинаково. Поскольку они практически не зависят от конкретного устройства, одна и та же функция может выводить данные как на жесткий диск, так и на консоль. Существуют два вида потоков: текстовый и бинарный.

Текстовые потоки

Текстовый поток представляет собой последовательность символов. Стандарт языка С позволяет (но не требует) организовывать потоки в виде строк, заканчивающихся символом перехода. В последней строке символ перехода указывать не обязательно. (На самом деле большинство компиляторов языка C/C++ не завершают текстовые потоки символом перехода на новую строку.) В зависимости от окружения некоторые символы в текстовых потоках могут подвергаться преобразованиям. Например, символ перехода на новую строку может быть заменен парой символов, состоящей из символа возврата каретки и прогона бумаги. Следовательно, между символами, записанными в текстовом потоке, и символами, выведенными на внешние устройства, нет взаимно однозначного соответствия. По этой же причине количество символов в текстовом потоке и на внешнем устройстве может быть разным.

Бинарные потоки

Бинарный поток — это последовательность байтов, однозначно соответствующая последовательности байтов, записанной на внешнем устройстве. Кроме того, количество записанных (или считанных) байтов совпадает с количеством байтов на внешнем устройстве. Однако бинарный поток может содержать дополнительные нулевые байты, количество которых зависит от конкретной реализации. Эти байты применяются для выравнивания записей, например для того, чтобы данные заполняли весь сектор на диске.

Файлы

В языке C/C++ *файлом* считается все — от файла на диске до дисплея или принтера. Выполнив операцию открытия, поток можно связать с конкретным файлом, который можно использовать для обмена данными с программой.

Не все файлы обладают одинаковыми возможностями. Например, файл на жестком диске предоставляет прямой доступ к своим записям, а некоторые принтеры — нет. Это приводит нас к следующему выводу: все потоки в файловой системе языка С одинаковы, а файлы могут различаться.

Если файл может поддерживать *запрос позиции* (position request), при его открытии *курсор файла* (file position indicator) устанавливается в начало. При чтении или записи очередного символа курсор перемещается на одну позицию вперед.

При закрытии файла его связь с потоком разрывается. Если файл был открыт для записи, его содержимое записывается на внешнее устройство. Этот процесс обычно называют *очисткой потока* (flushing). Он гарантирует, что после закрытия файла в потоке не останется никакой случайно забытой информации. При нормальном завершении программы все файлы закрываются автоматически. Если работа программы была завершена аварийно, например вследствие ошибки или выполнения функции `abort()`, файлы не закрываются.

Каждый поток, связанный с файлом, имеет управляющую структуру типа `FILE`, которую нельзя модифицировать.

Если вы новичок в программировании, различия между файлами и потоками могут показаться вам надуманными. Просто помните, что их единственное предназначение —

обеспечить унифицированный интерфейс. При выполнении операций ввода-вывода следует мыслить терминами потоков, используя при этом единственную файловую систему. Она автоматически преобразует исходные операции ввода или вывода, связанные с конкретным физическим устройством, в легко управляемый поток.

Основы файловой системы

Файловая система языка C состоит из нескольких взаимосвязанных функций. В табл. 9.1 приведены наиболее распространенные из них. Для их использования необходим заголовочный файл **stdio.h**. В программах на языке C++ можно также применять заголовочный файл **<cstdio>**.

Таблица 9.1. Наиболее распространенные функции ввода-вывода

Функция	*Операция*
fopen()	Открывает файл
fclose()	Закрывает файл
putc()	Записывает символ в файл
fputc()	То же, что и **putc()**
getc()	Считывает символ из файла
fgetc()	То же, что и **getc()**
fgets()	Считывает строку из файла
fputs()	То же, что и **gets()**
fseek()	Устанавливает курсор на заданный байт файла
ftell()	Возвращает текущую позицию курсора
fprintf()	Файловый аналог функции **printf()**
fscanf()	Файловый аналог функции **scanf()**
feof()	Возвращает истинное значение, если достигнут конец файла
ferror()	Возвращает истинное значение, если произошла ошибка
rewind()	Устанавливает курсор в начало файла
remove()	Стирает файл
fflush()	Очищает поток

Прототипы функций ввода-вывода находятся в заголовочном файле **stdio.h** (соответственно **<cstdio>**). Кроме того, в нем определены три типа: **size_t**, **fpos_t** и **FILE**. Тип **size_t**, как и **fpos_t**, представляет собой целочисленный тип без знака. Тип **FILE** обсуждается в следующем разделе.

В файле **stdio.h** (**<cstdio>**) определены макросы **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** и **SEEK_END**. Макрос **NULL** определяет нулевой указатель. Макрос **EOF** обычно определяет константу, равную –1. Он задает значение, возвращаемое функцией ввода при попытке прочесть несуществующую запись после конца файла. Макрос **FOPEN_MAX** определяет количество файлов, которые можно открыть одновременно. Остальные макросы используются функцией **fseek()**, осуществляющей прямой доступ к записям файла.

Указатель файла

Указатель файла — это звено, связывающее между собой все компоненты системы ввода-вывода. Он представляет собой указатель на структуру, имеющую тип **FILE**. В этой структуре хранится информация о файле, в частности, его имя, статус и текущее положение курсора. По существу, указатель файла описывает конкретный файл

и используется соответствующим потоком при выполнении операций ввода-вывода. Выполнить эти операции без указателя файла невозможно. Чтобы объявить указатель файла, следует выполнить следующий оператор.

```
FILE *fp;
```

Открытие файла

Функция **fopen()** открывает поток и связывает его с файлом. Затем она возвращает указатель на этот файл. Наиболее часто файлом считается физический файл, расположенный на диске. Именно такие файлы мы будем иметь в виду в ходе дальнейшего изложения. Прототип функции **fopen()** имеет следующий вид.

```
FILE *fopen(const char *имя_файла, const char *режим)
```

Здесь параметр *имя_файла* представляет собой указатель на строку символов, которая задает допустимое имя файла и может включать в себя описание пути к нему. Строка, на которую ссылается указатель *режим*, определяет предназначение файла. Допустимые значения параметра *режим* представлены в табл. 9.2.

Таблица 9.2. Допустимые значения параметра режим

Значение	*Смысл*
r	Открыть текстовый файл для чтения
w	Создать текстовый файл для записи
a	Добавить записи в конец текстового файла
rb	Открыть бинарный файл для чтения
wb	Создать бинарный файл для записи
ab	Добавить записи в конец бинарного файла
r+	Открыть текстовый файл для чтения и записи
w+	Создать текстовый файл для чтения и записи
a+	Добавить записи в конец текстового файла или создать текстовый файл для чтения и записи
r+b	Открыть бинарный файл для чтения и записи
w+b	Создать бинарный файл для чтения и записи
a+b	Добавить записи в конец бинарного файла или создать бинарный файл для чтения и записи

Как указывалось ранее, функция **fopen()** возвращает указатель файла. Не следует изменять значение этого указателя в собственной программе. Если во время открытия файла произойдет ошибка, функция **fopen()** вернет нулевой указатель.

В приведенном ниже примере функция **fopen()** используется для открытия файла с именем TEST.

```
FILE *fp;
fp = fopen("test", "w");
```

Этот фрагмент программы абсолютно верен, и все же файл можно открыть намного грамотнее. Например, так.

```
FILE *fp;
if ((fp = fopen("test","w"))==NULL){
  printf("Невозможно открыть файл.\n");
  exit(1);
}
```

Теперь можно обнаружить ошибку, возникшую при открытии файла, например, попытку открыть файл, защищенный от записи, или переполнение диска, и правильно

на нее отреагировать. В принципе, прежде чем приступать к выполнению операций ввода-вывода, следует всегда проверять, успешно ли выполнена функция **fopen()**.

Хотя большинство значений параметра *режим* имеют вполне очевидный смысл, некоторые из них требуют дополнительных разъяснений. Если при попытке открыть файл только для чтения выяснится, что такого файла на диске нет, функция **fopen()** вернет признак ошибки. Если попытаться открыть несуществующий файл для добавления записей, он будет создан. Кроме того, если для добавления открыть существующий файл, все новые записи будут записаны в его конец. Исходное содержание файла останется неизменным. Если несуществующий файл открывается для записи, он также создается. Если для записи открывается существующий файл, его содержимое уничтожается и заменяется новыми данными. Разница между режимами **r+** и **w+** заключается в том, что при попытке открыть несуществующий файл в режиме **r+** новый файл не создается, в отличие от режима **w+**. К тому же, если файл ранее существовал, то в режиме **w+** его содержимое будет уничтожено, а в режиме **r+** — нет.

Как следует из табл. 9.2, файл можно открыть как в текстовом, так и в бинарном режиме. Как правило, команды возврата каретки и прогона бумаги в текстовом режиме переводятся в символы перехода на новую строку. При выводе все происходит наоборот: символы перехода на новую строку интерпретируются как команды перевода каретки и прогона бумаги. При работе с бинарными файлами такие преобразования не выполняются.

Количество одновременно открытых файлов определяется константой **FOPEN_MAX**. Как правило, она равна 8, но ее точное значение следует искать в документации, сопровождающей компилятор.

Закрытие файла

Функция **fclose()** закрывает поток, открытый ранее функцией **fopen()**. Она записывает все оставшиеся в буфере данные в файл и закрывает его, используя команды операционной системы. Ошибка, возникшая при закрытии файла, может породить множество проблем, начиная с потери данных и разрушения файлов и заканчивая непредсказуемыми последствиями для программы. Кроме того, функция **fclose()** освобождает управляющий блок файла, связанного с потоком, позволяя использовать этот блок повторно. Операционная система ограничивает количество одновременно открытых файлов, поэтому, прежде чем открыть один файл, следует закрыть другой.

Прототип функции **fclose()** выглядит следующим образом.

```
int fclose(FILE *fp)
```

Здесь параметр *fp* представляет собой указатель файла, возвращенный функцией **fopen()**. Если функция вернула нулевой указатель, значит, файл закрыт успешно. Значение **EOF** является признаком ошибки. Для распознавания и обработки возникших проблем можно использовать стандартную функцию **ferror()**. Как правило, функция **fclose()** выполняется неверно, если диск был преждевременно вынут из дисковода или переполнен.

Запись символа

Для вывода символов предназначены две функции: **putc()** и **fputc()**. (На самом деле **putc()** — это макрос.) Эти эквивалентные функции просто сохраняют совместимость со старыми версиями языка C. В книге используется макрос **putc()**, хотя с таким же успехом можно было бы вызывать функцию **fputc()**.

Функция **putc()** записывает символ в файл, открытый с помощью функции **fopen()**. Прототип этой функции выглядит так.

```
int putc(int символ, FILE *fp)
```

Здесь параметр *fp* представляет собой указатель файла, возвращенный функцией *fopen()*, а аргумент *ch* является символом, подлежащим выводу. Указатель файла сообщает функции **putc()**, в какой именно файл следует произвести запись. Несмотря на то что параметр *ch* имеет тип **int**, в файл записывается лишь младший байт.

Если функция **putc()** выполнена успешно, она возвращает символ, записанный ею в файл. В противном случае она возвращает константу **EOF**.

Чтение символа

Ввод символа осуществляется двумя эквивалентными функциями: **getc()** и **fgetc()**. Наличие одинаковых функций позволяет сохранить совместимость со старыми версиями языка С. В дальнейшем мы будем использовать лишь функцию **getc()**, которая реализована в виде макроса. При желании можно применять и функцию **fgetc()**.

Функция **getc()** считывает символ из файла, открытого функцией **fopen()** в режиме чтения. Прототип функции **getc()** имеет следующий вид.

```
int getc(FILE *fp)
```

Здесь параметр *fp* является указателем файла, возвращенным функцией **fopen()**. Функция **getc()** возвращает целочисленную переменную, младший байт которой содержит введенный символ. Если возникла ошибка, старший байт этой переменной равен нулю.

Если при чтении обнаруживается конец файла, функция **getc()** возвращает константу **EOF**. Следовательно, для считывания данных вплоть до конца файла можно применять следующий фрагмент программы.

```
do {
  ch = getc(fp);
} while(ch!=EOF);
```

Однако функция **getc()** возвращает константу **EOF** и при возникновении ошибок. Для распознавания ошибки можно применять функцию **ferror()**.

Применение функций fopen(), getc(), putc() и fclose()

Функции **fopen()**, **getc()**, **putc()** и **fclose()** образуют минимальный набор процедур для работы с файлами. Приведенная ниже программа KTOD иллюстрирует применение функций **putc()**, **fopen()** и **fclose()**. Она считывает символы с клавиатуры и записывает их в файл, пока пользователь не введет символ **$**. Имя файла задается в командной строке. Например, вызвав программу KTOD с помощью команды **KTOD TEST**, можно записать текст в файл с именем **TEST**.

```
/* Программа KTOD для работы с диском. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Вы забыли указать имя файла.\n");
    exit(1);
  }
```

```
  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  do {
    ch = getchar();
    putc(ch, fp);
  } while (ch != '$');

  fclose(fp);

  return 0;
}
```

Программа DTOS выполняет противоположные операции: она считывает произвольный текстовый файл и выводит его содержимое на экран. Она демонстрирует использование функции **fgetc()**.

```
/* Программа DTOS считывает файл и выводит его на экран. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("Вы забыли ввести имя файла.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "r"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  ch = getc(fp);    /* Считываем символ */
  while (ch!=EOF) {
    putchar(ch);  /* Выводим его на экран */
    ch = getc(fp);
  }

  fclose(fp);
  return 0;
}
```

Чтобы выполнить тестирование этих программ, следует сначала создать текстовый файл с помощью программы KTOD, а затем считать его содержимое, используя программу DTOS.

Применение функции feof()

Как указано выше, обнаружив конец файла, функция **getc()** возвращает константу **EOF**. Однако это не самый лучший способ для распознавания конца файла. Во-первых, операционная система может работать как с текстовыми, так и с бинарными файлами. Если файл открыт для бинарного ввода, из него может быть считано целое

число, равное константе **EOF**. Следовательно, конец файла, распознаваемый функцией **getc()**, может не совпадать с реальным концом файла. Во-вторых, функция **getc()** возвращает константу **EOF** не только по достижении конца файла, но и при возникновении любой другой ошибки. По этой причине константу **EOF** невозможно интерпретировать однозначно. Для решения этой проблемы в языке С предусмотрена функция **feof()**, распознающая конец файла. Ее прототип имеет следующий вид.

```
int feof(FILE *fp)
```

Обнаружив конец файла, функция **feof()** возвращает истинное значение, в противном случае она возвращает число 0. Таким образом, приведенная ниже процедура считывает бинарный файл, пока не обнаружит его конец.

```
while(!feof(fp)) ch = getc(fp);
```

Разумеется, этот способ можно применять как для бинарных, так и для текстовых файлов.

Следующая программа, копирующая текстовые или бинарные файлы, иллюстрирует применение функции **feof()**.

```
/* Копирование файла. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("Вы забыли ввести имя файла.\n");
  exit(1);
  }

  if((in=fopen(argv[1], "rb"))==NULL) {
    printf("Невозможно открыть исходный файл.\n");
    exit(1);
  }
    if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Невозможно открыть результирующий файл\n");
    exit(1);
  }

  /* Копирование файла. */
  while(!feof(in)) {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

Работа со строками: функции fputs() и fgets()

Кроме функций **getc()** и **putc()** файловая система языка С содержит функции **fgets()** и **fputs()**, выполняющие чтение символьных строк из файла и запись их в файл соответственно. Эти функции аналогичны функциям **getc()** и **putc()**, однако они считывают и записывают строки, а не отдельные символы. Вот как выглядят их прототипы.

```
int fputs(const char *строка, FILE *fp);
char *fgets(char *строка, int длина, FILE *fp);
```

Функция **fputs()** записывает в заданный поток строку, на которую ссылается указатель *строка*. При возникновении ошибки она возвращает константу **EOF**.

Функция **fgets()** считывает строку из указанного потока, пока не обнаружит символ перехода или не прочитает *длина*–1 символов. В отличие от функции **getc()** символ перехода считается составной частью строки. Результирующая строка должна завершаться нулем. В случае успеха функция возвращает указатель на введенную строку, в противном случае она возвращает нулевой указатель.

Следующая программа иллюстрирует работу функции **fputs()**. Она считывает строку с клавиатуры и записывает ее с файл с именем TEST. Для прекращения работы программы следует ввести пустую строку. Поскольку функция **gets()** не позволяет вводить символ перехода, при записи в файл он явно дописывается перед каждой строкой. Это облегчает последующее чтение файла.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  do {
    printf("Введите строку (для выхода нажмите клавишу <ENTER>):\n");
    gets(str);
    strcat(str, "\n");  /* Добавляем символ перехода */
    fputs(str, fp);
  } while(*str!='\n');

  return 0;
}
```

Функция rewind()

Функция **rewind()** устанавливает курсор в начало файла, заданного в качестве ее аргумента. Иными словами, она "перематывает" (rewind) файл в начало. Ее прототип имеет следующий вид.

```
void rewind(FILE *fp);
```

Здесь параметр *fp* является допустимым указателем файла.

Продемонстрируем работу функции **rewind()**, модифицировав программу из предыдущего раздела так, чтобы она отображала на экране содержимое только что созданного файла. Для этого программа после считывания данных устанавливает курсор файла в начало, а затем использует функцию **fgets()** для повторного считывания. Обратите внимание на то, что теперь файл должен быть открыт в режиме чтения-записи с помощью параметра **w+**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  char str[80];
  FILE *fp;

  if((fp = fopen("TEST", "w+"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  do {
    printf("Введите строку (для выхода нажмите клавишу <ENTER>):\n");
    gets(str);
    strcat(str, "\n");  /* Добавляем символ перехода */
    fputs(str, fp);
  } while(*str!='\n');

  /* Теперь считываем файл и отображаем его на экране */
  rewind(fp);  /* Устанавливаем курсор файла в начало. */
  while(!feof(fp)) {
    fgets(str, 79, fp);
    printf(str);
  }

  return 0;
}
```

Функция ferror()

Функция **ferror()** определяет, возникла ли ошибка при работе с файлом. Ее прототип имеет следующий вид.

```
int ferror(FILE *fp);
```

Здесь параметр *fp* является допустимым указателем файла. Функция возвращает истинное значение, если при выполнении операции возникла ошибка, в противном случае она возвращает ложное значение. Поскольку с каждой операцией над файлом связан определенный набор ошибок, функцию **ferror()** следует вызывать сразу же после выполнения операции, в противном случае ошибка может быть упущена.

Следующая программа иллюстрирует работу функции **ferror()**. Она удаляет из файла символы табуляции, заменяя их соответствующим количеством пробелов, которое задается константой **TAB_SIZE**. Обратите внимание на то, что функция **ferror()** вызывается после выполнения каждой файловой операции. При запуске программы укажите в командной строке имена входного и выходного файлов.

```
/* Программа заменяет символы табуляции в текстовом файле
   соответствующим количеством пробелов и проверяет ошибки */
#include <stdio.h>
#include <stdlib.h>

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e);
int main(int argc, char *argv[])
{
  FILE *in, *out;
  int tab, i;
  char ch;

  if(argc!=3) {
    printf("usage: detab <in> <out>\n");
    exit(1);
  }

  if((in = fopen(argv[1], "rb"))==NULL) {
    printf("Невозможно открыть файл %s.\n", argv[1]);
    exit(1);
  }

  if((out = fopen(argv[2], "wb"))==NULL) {
    printf("Невозможно открыть файл %s.\n", argv[1]);
    exit(1);
  }

  tab = 0;
  do {
    ch = getc(in);
    if(ferror(in)) err(IN);

    /* Символ табуляции заменяется соответствующим количеством пробелов */
    if(ch=='\t') {
      for(i=tab; i<8; i++) {
       putc(' ', out);
       if(ferror(out)) err(OUT);
      }
      tab = 0;
    }
    else {
      putc(ch, out);
      if(ferror(out)) err(OUT);
      tab++;
      if(tab==TAB_SIZE) tab = 0;
      if(ch=='\n' || ch=='\r') tab = 0;
    }
  } while(!feof(in));
  fclose(in);
  fclose(out);

  return 0;
}
```

```
void err(int e)
{
  if(e==IN) printf("Ошибка при вводе.\n");
  else printf("Ошибка при выводе.\n");
  exit(1);
}
```

Удаление файла

Функция **remove()** удаляет указанный файл. Ее прототип имеет следующий вид.

```
int remove(const char *имя_файла)
```

Если функция выполнена успешно, она возвращает нуль, в противном случае она возвращает ненулевое значение.

Следующая программа удаляет файл, заданный в командной строке. Однако сначала она дает пользователю возможность передумать. Такие утилиты могут быть полезными для новичков.

```
/* Двойная проверка перед удалением файла. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
  char str[80];
  if(argc!=2) {
    printf("Командная строка: xerase <filename>\n");
    exit(1);
  }

  printf("Стереть %s? (Y/N): ", argv[1]);
  gets(str);

  if(toupper(*str)=='Y')
    if(remove(argv[1])) {
      printf("Невозможно удалить файл.\n");
      exit(1);
    }
  return 0;
}
```

Очистка потока

Для того чтобы очистить поток вывода, следует применять функцию **fflush()**, имеющую следующий прототип.

```
int fflush(FILE *fp);
```

Эта функция записывает содержимое буфера в файл, связанный с указателем *fp*. Если вызвать функцию **fflush()** с нулевым аргументом, данные, содержащиеся в буферах, будут записаны во все файлы, открытые для записи.

В случае успешного выполнения функция **fflush()** возвращает 0, в противном случае она возвращает константу **EOF**.

Функции fread() и fwrite()

Для чтения и записи данных, размер типа которых превышает 1 байт, файловая система языка С содержит две функции: **fread()** и **fwrite()**. Эти функции позволяют считывать и записывать блоки данных любого типа. Их прототипы имеют следующий вид.

```
size_t fread(void *буфер, size_t количество_байтов,
             size_t количество_блоков, FILE *fp)
size_t fwrite(const void *буфер, size_t количество_байтов,
             size_t количество_блоков, FILE *fp)
```

В прототипе функции **fopen()** параметр *буфер* представляет собой указатель на область памяти, в которую записываются данные, считанные из файла. В прототипе функции **fwrite()** параметр *буфер* представляет собой указатель на область памяти, содержащую данные, которые должны быть записаны в файл. Значение параметра *количество_блоков* определяет, сколько блоков подлежит считыванию или записи. Длина блоков задается параметром *количество_байтов*. (Напомним, что тип **size_t** определен как целое число без знака.) В заключение отметим, что указатель *fp* является указателем на ранее открытый поток.

Функция **fread()** возвращает количество считанных блоков. Оно может быть меньше параметра *количество_блоков*, если обнаружен конец файла или возникла ошибка. Функция **fwrite()** возвращает количество записанных блоков. Если не произошло никаких ошибок, это значение равно параметру *количество_блоков*.

Применение функций fread() и fwrite()

Если файл был открыт в бинарном режиме, его чтение и запись можно осуществлять с помощью функций **fread()** и **fwrite()**. Например, следующая программа сначала записывает в файл, а затем считывает обратно переменные типов **double**, **int** и **long**. Обратите внимание на то, что в программе используется оператор **sizeof**, определяющий размер каждого типа данных.

```
/* Записываем данные, не являющиеся символами, на диск
   и считываем их обратно. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  double d = 12.23;
  int i = 101;
  long l = 123023L;

  if((fp=fopen("test", "wb+"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  fwrite(&d, sizeof(double), 1, fp);
  fwrite(&i, sizeof(int), 1, fp);
  fwrite(&l, sizeof(long), 1, fp);

  rewind(fp);
  fread(&d, sizeof(double), 1, fp);
```

```
  fread(&i, sizeof(int), 1, fp);
  fread(&l, sizeof(long), 1, fp);

  printf("%f %d %ld", d, i, l);
  fclose(fp);
  return 0;
}
```

Буфер, как правило, представляет собой обычную область памяти, предназначенную для хранения переменных. В данной простой программе значения, возвращаемые функциями **fread()** и **fwrite()**, игнорируются. Однако в реальных программах эти значения следует анализировать, распознавая возможные ошибки.

В большинстве случаев функции **fread()** и **fwrite()** используются для считывания сложных данных, тип которых определен пользователем, особенно структур. Например, допустим, что в программе определена следующая структура.

```
struct struct_type {
  float balance;
  char name[80];
} cust;
```

Следующий оператор записывает содержимое переменной **cust** в файл, на который ссылается указатель **fp**.

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

Функция fseek() и файлы с произвольным доступом

С помощью функции **fseek()** можно осуществлять операции ввода и вывода данных, используя файлы с произвольным доступом. Эта функция устанавливает курсор файла в заданную позицию. Ее прототип имеет следующий вид.

```
int fseek(FILE *fp, long int смещение, int начало);
```

Здесь параметр *fp* представляет собой указатель файла, возвращенный функцией **fopen()**. Параметр *смещение* указывает количество байтов, на которое следует переместить курсор файла от точки, заданной параметром *начало*. Значение параметра *начало* задается одним из следующих макросов.

Начало отсчета	*Имя макроса*
Начало файла	**SEEK_SET**
Текущая позиция	**SEEK_CUR**
Конец файла	**SEEK_END**

Таком образом, чтобы переместить курсор файла, находящийся в начале файла, на количество байтов, заданное параметром *смещение*, следует применять макрос **SEEK_SET**. Если точкой отсчета является текущая позиция курсора, используется макрос **SEEK_CUR**, а если курсор файла установлен на последнюю запись, следует применять макрос **SEEK_END**. Если функция **fseek()** выполнена успешно, она возвращает значение 0, а если возникла ошибка, возвращается ненулевое значение.

Следующая программа иллюстрирует применение функции **fseek()**. Она перемещает курсор в заданную позицию указанного файла и отображает байт, записанный в этом месте. Имя файла задается в командной строке.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  if(argc!=3) {
    printf("Командная строка: SEEK имя_файла количество_байтов\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  if(fseek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Ошибка.\n");
    exit(1);
  }

  printf("В позиции %ld записан символ %c.\n", atol(argv[2]), getc(fp));
  fclose(fp);

  return 0;
}
```

Функцию **fseek()** можно применять для перемещения на заданное количество байтов, умножая размер соответствующего типа на количество элементов. Допустим, например, что список рассылки состоит из структур, имеющих тип **list_type**. Для перехода на десятый адрес, записанный в файле, используется следующий оператор.

```
fseek(fp, 9*sizeof(struct list_type), SEEK_SET);
```

Текущую позицию файлового курсора можно определить с помощью функции **ftell()**. Ее прототип имеет следующий вид.

```
long int ftell(FILE *fp);
```

Данная функция возвращает текущее положение курсора файла, связанного с указателем *fp*. Если возникла ошибка, она возвращает значение −1.

Как правило, произвольный доступ нужен лишь при работе с бинарными файлами. Причина довольно проста. Поскольку текстовые файлы могут содержать управляющие символы, которые интерпретируются как команды, количество байтов, на которое следует выполнить смещение курсора, может не соответствовать ожидаемому символу. Единственная ситуация, в которой функцию **fseek()** можно применять к текстовым файлам, возникает, лишь если позиция курсора была ранее определена с помощью функции **ftell()** и макроса **SEEK_SET**.

Следует помнить одну важную вещь: файл, содержащий текст, можно открыть в бинарном режиме. В этом случае на него не распространяются ограничения, касающиеся произвольного доступа к записям текстового файла. Эти ограничения относятся лишь к файлам, открытым в текстовом режиме.

Функции fprintf() и fscanf()

Кроме уже упомянутых функций ввода-вывода, в языке С предусмотрены функции **fprintf()** и **fscanf()**. Эти функции совершенно аналогичны функциям **printf()** и **scanf()**, за исключением одного — они работают с файлами. Прототипы функций **fprintf()** и **fscanf()** имеют следующий вид.

```
int fprintf(FILE *fp, const char *управляющая_строка, ...);
int fscanf(FILE *fp, const char *управляющая_строка, ...);
```

Здесь параметр *fp* представляет собой указатель файла, возвращенный функцией **fopen()**. Функции **fprintf()** и **fscanf()** выполняют операции ввода-вывода с файлом, на который ссылается указатель *fp*.

Рассмотрим в качестве примера программу, считывающую с клавиатуры строку и целое число, а затем записывающую эту информацию в файл с именем TEST. Затем программа читает этот файл и выводит считанные данные на экран. Проверьте содержимое файла TEST после выполнения программы и убедитесь, что его можно понять.

```
/* Пример использования функций fscanf() и fprintf() */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>

int main(void)
{
  FILE *fp;
  char s[80];
  int t;

  if((fp=fopen("test", "w")) == NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  printf("введите строку и целое число: ");
  fscanf(stdin, "%s%d", s, &t); /* Считываем данные с клавиатуры */

  fprintf(fp, "%s %d", s, t); /* Записываем их в файл */
  fclose(fp);

  if((fp=fopen("test","r")) == NULL) {
    printf("Невозможно открыть файл.\n");
    exit(1);
  }

  fscanf(fp, "%s%d", s, &t); /* Считываем данные из файла */
  fprintf(stdout, "%s %d", s, t); /* Выводим их на экран */

  return 0;
}
```

Учтите: хотя функции **fprintf()** и **fscanf()** часто обеспечивают наиболее простой способ чтения и записи разнообразных данных, они не всегда эффективны. Поскольку данные, записанные в файле с помощью формата ASCII, точно соответствуют своему представлению на экране, каждый вызов этих функций сопряжен с дополнительными расходами. Таким образом, если скорость выполнения операций ввода-вывода и размер файла являются критическими параметрами, следует применять функции **fread()** и **fwrite()**.

Стандартные потоки

В начале выполнения любой программы, написанной на языке C, автоматически открываются три потока: **stdin** (стандартный поток ввода), **stdout** (стандартный поток вывода) и **stderr** (стандартный поток ошибок). Обычно эти потоки связаны с консолью, однако операционная система может перенаправлять их на другие устройства. (Перенаправляемые операции ввода-вывода поддерживаются, например, операционными системами Windows, DOS, Unix и OS/2.)

Поскольку стандартные потоки представляют собой указатели файлов, их можно использовать для консольного ввода-вывода. Например, функцию **putchar()** можно определить следующим образом.

```
int putchar(char c)
{
   return putc(c, stdout);
}
```

Как правило, поток **stdin** используется для ввода с клавиатуры, а потоки **stdout** и **stderr** используются для записи на экран дисплея.

Потоки **stdin**, **stdout** и **stderr** можно применять как файловые указатели в любой функции, использующей указатели типа **FILE***. Например, функцию **fgets()** можно применять для ввода строки с консоли, используя следующий вызов.

```
char str[255];
fgets(str, 80, stdin);
```

Фактически такое использование функции **fgets()** довольно полезно. Как упоминалось ранее, применение функции **gets()** может привести к переполнению массива, в который записываются символы, поскольку она не предусматривает проверки возможного выхода индекса массива за пределы допустимого диапазона. При использовании потока **stdin** функция **fgets()** представляет собой разумную альтернативу, поскольку она выполняет проверку индекса массива и предотвращает его переполнение. Остается лишь одно неудобство: функция **fgets()**, в отличие от функции **gets()**, не удаляет из файла символ перехода на новую строку, поэтому его необходимо удалять вручную, как показано в следующем примере.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80];
  int i;

  printf("Введите строку: ");
  fgets(str, 10, stdin);

  /* Удаляем символ перехода на новую строку, если он есть */
  i = strlen(str)-1;
  if(str[i]=='\n') str[i] = '\0';

  printf("Результат: %s", str);
  return 0;
}
```

Имейте в виду, что потоки **stdin**, **stdout** и **stderr** не являются переменными в общепринятом смысле этого слова и им ничего нельзя присвоить с помощью функции **fopen()**. Кроме того, они открываются и закрываются автоматически, поэтому не следует пытаться их закрывать самостоятельно.

Связь с консольным вводом-выводом

Между консольным и файловым вводом-выводом существует небольшое различие. Функции консольного ввода-вывода, описанные в главе 8, работают с потоками **stdin** или **stdout**. По существу, функции консольного ввода-вывода просто являются специальными версиями своих файловых аналогов и включены в язык лишь для удобства программистов.

Как описано в предыдущем разделе, консольный ввод-вывод можно осуществлять с помощью функций файловой системы. Однако для вас может оказаться сюрпризом, что файловые операции ввода-вывода можно выполнять с помощью консольных функций, например функции **printf()**! Это возможно благодаря тому, что консольные функции ввода-вывода работают с потоками **stdin** и **stdout**. Если операционная среда позволяет перенаправлять потоки, потоки **stdin** и **stdout** можно связать не с клавиатурой и экраном, а с другими устройствами. Рассмотрим следующую программу.

```
#include <stdio.h>

int main(void)
{
  char str[80];
  printf("Введите строку: ");
  gets(str);
  printf(str);

  return 0;
}
```

Предположим, что эта программа называется TEST. При нормальном выполнении эта программа выводит на экран приглашение, считывает с клавиатуры строку и выводит ее на экран. Однако в среде, допускающей перенаправление потоков, стандартные потоки **stdin** и **stdout** можно связать с файлами. Например, в операционных системах DOS и Windows программу TEST можно выполнить с помощью следующей командной строки.

```
TEST > OUTPUT
```

В этом случае результаты работы **TEST** будут записаны в файл **OUTPUT**. Если для запуска программы применить командную строку

```
TEST < INPUT > OUTPUT,
```

поток **stdin** будет связан с файлом **INPUT**, а результаты — направлены в файл **OUTPUT**.

После завершения работы программы потоки возвращаются в стандартное состояние.

Применение функции freopen() для перенаправления стандартных потоков

Для перенаправления стандартных потоков можно применять функцию **freopen()**. Эта функция связывает существующий поток с новым файлом. Таким образом, с ее помощью стандартный поток можно связать с новым файлом. Ее прототип имеет следующий вид.

```
FILE *freopen(const char *имя_файла, const char *режим, FILE *поток)
```

Здесь параметр *имя_файла* является указателем на строку, содержащую имя файла, связанного с заданным *потоком*. Файл открывается в режиме, заданном параметром *режим*, который может иметь те же значения, что и параметр *режим* в функции **fopen()**. В случае успешного выполнения функция **freopen()** возвращает указатель файла *поток*, в противном случае она возвращает нулевой указатель.

В следующей программе функция **freopen()** перенаправляет стандартный поток **stdout** в файл **OUTPUT**.

```
#include <stdio.h>

int main(void)
{
  char str[80];
  freopen("OUTPUT", "w", stdout);
  printf("Введите строку: ");
  gets(str);
  printf(str);

  return 0;
}
```

Как правило, перенаправление стандартных потоков с помощью функции **freopen()** оказывается полезным в особых ситуациях, например, при отладке. Однако этот способ менее эффективен, чем применение функций **fread()** и **fwrite()**.

Полный справочник по

Глава 10

Препроцессор и комментарии

В программы на языке C/C++ можно включать различные инструкции, регламентирующие работу компилятора. Эти инструкции называются *директивами препроцессора* (preprocessor directives). Хотя они не являются частью языков C или C++, их применение позволяет расширить возможности программ. Кроме этих директив, в главе рассматриваются комментарии.

Препроцессор

Прежде чем приступить к описанию препроцессора, вспомним его историю. Препроцессор языка C++ унаследован от языка C. Более того, он практически совпадает с препроцессором языка C. Единственное различие между ними заключается в степени их важности. В языке C каждая директива препроцессора является необходимой. В языке C++ некоторые избыточные директивы компенсируются элементами самого языка. Фактически одной из долговременных целей языка C++ является полное исключение препроцессора. Однако в настоящее время директивы препроцессора применяются очень часто, и в обозримом будущем это положение не изменится.

Препроцессор содержит следующие директивы.

#define	**#elif**	**#else**	**#endif**
#error	**#if**	**#ifdef**	**#ifndef**
#include	**#line**	**#pragma**	**#undef**

Все директивы препроцессора начинаются со знака **#**. Кроме того, каждая директива должна располагаться в отдельной строке программы. Например, фрагмент программы, приведенный ниже, неверен.

```
#include <stdio.h>  #include <stdlib.h>
```

Директива #define

Директива **#define** определяет идентификатор и последовательность символов, которая заменяет его в тексте программы. Этот идентификатор называется *именем макроса*, а процесс замены — *макроподстановкой* (macro replacement). Общий вид этой директивы таков.

#define *имя_макроса последовательность_символов*

Обратите внимание на то, что эта директива не содержит точки с запятой. Между идентификатором и последовательностью символов может располагаться сколько угодно пробелов, но сама последовательность должна заканчиваться символом перехода на новую строку.

Например, если вы хотите использовать вместо единицы слово **LEFT**, а вместо нуля — слово **RIGHT**, необходимо объявить две директивы **#define**.

```
#define LEFT 1
#define RIGHT 0
```

Эти директивы вынудят компилятор заменять слова **LEFT** и **RIGHT** в исходном файле значениями 1 и 0 соответственно. Например, следующий оператор выводит на экран числа **0 1 2**.

```
printf("%d %d %d", RIGHT, LEFT, LEFT+1);
```

Если в программе определено имя макроса, его можно использовать для определения другого макроса. Например, в приведенном ниже фрагменте программы определяются значения идентификаторов **ONE**, **TWO** и **THREE**.

```
#define ONE    1
#define TWO    ONE+ONE
#define THREE  ONE+TWO
```

Макроподстановка означает простую замену идентификатора последовательностью символов, связанной с ним. Следовательно, при необходимости с помощью директивы **#define** можно определить стандартное сообщение об ошибке

```
#define E_MS "стандартная ошибка при вводе\n"
/* ... */
printf(E_MS);
```

Каждый раз, когда в исходном тексте программы встретится идентификатор **E_MS**, компилятор подставит вместо него строку "стандартная ошибка при вводе\n". С точки зрения компилятора, приведенный выше вызов функции **printf()** выглядит следующим образом.

```
printf("стандартная ошибка при вводе\n");
```

Если идентификатор является частью строки, заключенной в кавычки, макроподстановка не производится. Например, фрагмент программы

```
#define XYZ это проверка

printf("XYZ");
```

выведет на экран не строку "**это проверка**", а символы **XYZ**.

Если последовательность символов занимает несколько строк, в конце каждой из них следует поставить обратную косую черту.

```
#define LONG_STRING "в этом примере \
используется очень длинная строка"
```

Для названия идентификаторов программисты обычно используют прописные буквы. Это позволяет легко обнаруживать макроподстановки в тексте программ. Кроме того, лучше помещать все директивы **#define** в самом начале исходного файла или в отдельном заголовочном файле, а не разбрасывать их по всей программе.

Макросы часто применяются для определения имен констант, встречающихся в программе. Допустим, в программе определен массив, который используется в нескольких модулях. Крайне нежелательно "зашивать" его размер в текст программы. Вместо этого следует применить макроподстановку, используя директиву **#define**. Тогда для того, чтобы изменить размер массива, достаточно будет модифицировать одну строку программы, содержащую директиву **#define**, а не выискивать все ссылки на размер массива в тексте. После этого программу необходимо перекомпилировать. Рассмотрим пример.

```
#define MAX_SIZE 100
/* ... */
float balance[MAX_SIZE];
/* ... */
for(i=0; i<MAX_SIZE; i++) printf("%f", balance[i]);
/* ... */
for(i=0; i<MAX_SIZE; i++) x =+ balance[i];
```

Поскольку размер массива **balance** определяется идентификатором **MAX_SIZE**, то для того, чтобы определить новый размер массива, достаточно просто модифицировать

определение макроса. При повторной компиляции программы все последующие ссылки на этот идентификатор будут обновлены автоматически.

На заметку *В языке C++ есть более простой способ определения констант, основанный на применении ключевого слова* **const**. *Он описан в части II.*

Определение функций в виде макросов

Директива **#define** обладает еще одним мощным свойством: макрос может иметь формальные аргументы. Каждый раз, когда в тексте программы встречается имя макроса, его формальные аргументы заменяются фактическими. Этот вид макроса называется *функциональным* (function-like macro). Рассмотрим пример.

```
#include <stdio.h>

#define ABS(a)   (a)<0 ? -(a) : (a)

int main(void)
{
  printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));

  return 0;
}
```

При компиляции программы определение макроса **a** будет заменено значениями −1 и 1. Скобки, в которые заключен макрос **a**, гарантируют правильную подстановку. Посмотрим, что произойдет, если убрать скобки вокруг макроса **a**. В этом случае выражение

```
ABS(10-20)
```

после макроподстановки будет преобразовано в следующий оператор.

```
10-20<0 ? -10-20 : 10-20
```

Очевидно, что результат этого выражения будет ошибочным.

Использование функциональных макросов вместо настоящих функций увеличивает скорость выполнения программы, поскольку в ней отсутствуют вызовы функций. Однако, если размер функционального макроса достаточно велик, быстродействие программы достигается в ущерб ее размеру, поскольку многие фрагменты программы просто дублируются.

На заметку *Несмотря на то что параметризованные макросы довольно полезны, в языке C++ есть более эффективный способ создания подставляемого кода, основанный на использовании ключевого слова* **inline**.

Директива #error

Директива **#error** вынуждает компилятор прекратить компиляцию. Она используется в основном при отладке программ. Общий вид директивы **#error** таков.

```
#error сообщение_об_ошибке
```

Параметр *сообщение_об_ошибке* представляет собой строку, не заключенную в кавычки. При выполнении директивы **#error** на экран выводится сообщение об ошибке, которое может сопровождаться другой информацией, определенной компилятором.

Директива #include

Директива **#include** вынуждает компилятор считать и подставить в исходный текст программы файл с заданным именем. Это имя заключается в двойные кавычки или угловые скобки. Например, директивы

```
#include "stdio.h"
#include <stdio.h>
```

заставляют компилятор считать и скомпилировать заголовочные файлы стандартных библиотечных функций ввода-вывода.

Включаемые файлы сами могут содержать директивы **#include**. Такие директивы называются *вложенными*. Глубина вложения директив зависит от компилятора. Стандарт языка C допускает восемь уровней вложения, а стандарт языка C++ — по крайней мере 256.

Кавычки и угловые скобки, в которых указываются имена включаемых файлов, определяют способ их поиска на жестком диске. Если имя файла содержится в угловых скобках, он должен находиться в каталоге, указанном компилятором. Обычно это каталог **INCLUDE**, предназначенный для хранения всех заголовочных файлов. Если имя файла заключено в кавычки, как правило, его поиск выполняется в рабочем каталоге. Если файл не найден, поиск повторяется так, будто имя файла содержалось в угловых скобках.

Большинство программистов заключают в угловые скобки имена заголовочных файлов, а кавычки оставляют для файлов, определенных пользователем. Однако на этот счет никаких жестких правил не предусмотрено. Каждый программист волен поступать так, как ему удобно.

Программы на языке C++ могут использовать директиву **#include** для включения не только *файлов*, но и *заголовков* (headers). В языке C++ предусмотрен набор стандартных заголовков, содержащих информацию, необходимую для работы различных библиотек. Заголовок представляет собой стандартный идентификатор, который может, но не обязан, являться именем файла. Таким образом, заголовок в языке C++ — это абстракция, гарантирующая, что в программу будет включена необходимая информация. Различные вопросы, связанные с использованием заголовков, обсуждаются в части II.

Директивы условной компиляции

Препроцессор содержит несколько директив, позволяющих выборочно компилировать отдельные части программы. Этот процесс называется *условной компиляцией* (conditional compilation). Он широко используется в коммерческих приложениях, поскольку позволяет настраивать их на конкретное окружение.

Директивы #if, #else, #elif и #endif

Вероятно, наиболее распространенными директивами условной компиляции являются директивы **#if**, **#else**, **#elif** и **#endif**. Эти директивы позволяют выбирать компилируемые части программы в зависимости от значения константного выражения.

Директива **#if** имеет следующий общий вид.

```
#if константное_выражение
    последовательность операторов
#endif
```

Если значение константного выражения истинно, код, заключенный между директивами **#if** и **#endif**, компилируется. В противном случае эта часть программы игнорируется. Директива **#endif** отмечает конец блока директивы **#if**. Рассмотрим пример.

```
/* Простой пример директивы #if. */
#include <stdio.h>

#define MAX 100

int main(void)
{
#if MAX>99
  printf("Компилируется, если размер массива больше  99.\n");
#endif

  return 0;
}
```

Эта программа выводит на экран сообщение, поскольку значение именованной константы **MAX** больше 99. Этот пример иллюстрирует один очень важный момент. Выражение, стоящее после директивы **#if**, вычисляется на этапе компиляции. Таким образом, оно может содержать лишь ранее определенные идентификаторы и константы, но не переменные.

Директива **#else** идентична оператору **else**, являющемуся частью языка C++: она определяет альтернативу директиве **#if**. Предыдущий пример можно немного усовершенствовать.

```
/* Simple #if/#else example. */
#include <stdio.h>

#define MAX 10

int main(void)
{
#if MAX>99
  printf("Компилируется, если размер массива больше 99.\n");
#else
  printf("Компилируется, если размер массива мал.\n");
#endif

  return 0;
}
```

В данном случае значение именованной константы **MAX** меньше 99, поэтому часть программы, указанная после директивы **#if**, не компилируется. Вместо нее компилируется альтернатива, связанная с директивой **#else**, и на экране появляется сообщение "**Компилируется, если размер массива мал**".

Обратите внимание на то, что директива **#else** одновременно является концом блока **#if** и началом альтернативного блока, поскольку директива **#if** может иметь только одну альтернативу **#else**.

Директива **#elif** означает "else if" и создает цепочку **if-else-if**, позволяя выполнять многовариантную условную компиляцию. Если выражение истинно, компилируется связанный с ним блок кода, а остальные выражения **#elif** не вычисляются. В противном случае проверяется следующий блок. Общий вид директивы **#elif** таков.

```
#if выражение
    последовательность операторов
#elif выражение 1
    последовательность операторов
#elif выражение 2
    последовательность операторов
#elif выражение 3
    последовательность операторов
#elif выражение 4
.
.
.
#elif выражение N
    последовательность операторов
#endif
```

Например, следующий фрагмент использует значение именованной константы **ACTIVE_COUNTRY** для определения знака, обозначающего валюту.

```
#define US 0
#define ENGLAND 1
#define FRANCE 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
  char currency[] = "доллар";
#elif ACTIVE_COUNTRY == ENGLAND
  char currency[] = "фунт";
#else
  char currency[] = "франк";
#endif
```

Стандарт языка C устанавливает, что директивы **#if** и **#elif** могут быть вложенными, а глубина вложения может достигать по крайней мере восьми уровней. Стандарт языка C++ допускает 256 уровней вложения. Вложенные директивы **#endif**, **#else** или **#elif** связываются с ближайшими директивами **#if** или **#elif**. Например, следующий фрагмент абсолютно верен.

```
#if MAX>100
  #if SERIAL_VERSION
    int port=198;
  #elif
    int port=200;
  #endif
#else
  char out_buffer[100];
#endif
```

Директивы #ifdef и ifndef

Другой способ условной компиляции основан на применении директив **#ifdef** и **#ifndef**, означающих "if defined" (если определено) и "if not defined" (если не определено). Общий вид директивы **#ifdef** таков.

```
#ifdef имя_макроса
    последовательность операторов
#endif
```

Если имя макроса ранее определено с помощью директивы **#define**, компилируется соответствующий блок кода.

Директива **#ifndef** имеет следующий общий вид.

```
#ifndef имя_макроса
    последовательность операторов
#endif
```

Если имя макроса не было определено с помощью директивы **#define**, компилируется соответствующий блок кода.

Директивы **#ifdef** и **#ifndef** могут использовать директивы **#else** и **#elif**. Например, следующий фрагмент программы выводит на экран строки "**Привет, Тед**" и "**Ральфа нет дома**". Однако, если именованная константа **TED** не определена, на экран выводится сообщение "**Всем привет**", а затем — "**Ральфа нет дома**".

```
#include <stdio.h>

#define TED 10

int main(void)
{
#ifdef TED
  printf("Привет, Тед\n");
#else
  printf("Всем привет\n");
#endif
#ifndef RALPH
  printf("Ральфа нет дома\n");
#endif

  return 0;
}
```

Директивы **#ifdef** и **#ifndef** могут быть вложенными, причем стандарт языка C допускает как минимум восемь уровней вложения, а стандарт языка C++ — до 256.

Директива #undef

Директива **#undef** удаляет определение указанного имени макроса. Иными словами, эта директива делает макрос неопределенным. Общий вид директивы **#undef** таков.

```
#undef имя_макроса
```

Рассмотрим, например, следующий фрагмент программы.

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
```

```
/* В этой точке именованные константы LEN и WIDTH
   не определены */
```

Именованные константы **LEN** и **WIDTH** считаются определенными, пока не будет выполнена директива **#undef**.

Оператор defined

Кроме директивы **#ifdef**, есть еще один способ проверить, определено ли имя макроса. Для этого можно применять директиву **#if** в сочетании со статическим оператором **defined**, выполняемым на этапе компиляции. Оператор **defined** имеет следующий общий вид.

```
defined имя_макроса
```

Если в данный момент имя макроса является определенным, выражение имеет истинное значение. В противном случае оно ложно. Например, чтобы распознать, является ли именованная константа **MYFILE** определенной, можно использовать одну из следуюших команд препроцессора:

```
#if defined MYFILE
```

или

```
#ifdef MYFILE
```

Поставив перед оператором **defined** символ **!**, можно задать противоположное условие. Например, следующий фрагмент программы компилируется только в том случае, если именованная константа **DEBUG** не определена.

```
#if !defined DEBUG
  printf("Final version!\n");
#endif
```

Одна из причин, по которой используется оператор **defined**, состоит в том, что он позволяет проверить, определено ли имя макроса с помощью оператора **#elif**.

Директива #line

Директива **#line** соответственно изменяет содержимое макросов _ _**LINE**_ _ и _ _**FILE**_ _, представляющих собой идентификаторы, определенные компилятором. Идентификатор _ _**LINE**_ _ содержит номер текущей строки компилируемого кода. Идентификатор _ _**FILE**_ _ представляет собой строку, содержащую имя компилируемого файла. Общий вид директивы **#line** таков.

#line *номер* "*имя файла*"

Здесь параметр *номер* равен любому положительному числу, которое становится новым значением макроса _ _**LINE**_ _. Параметр *имя файла* не обязателен и может быть любым допустимым именем файла, которое становится новым значением макроса _ _**FILE**_ _. Директива **#line** в основном используется при отладке программ и в специальных приложениях.

Например, следующий код устанавливает, что нумерацию строк следует начинать с числа 100. Функция **printf()** выводит на экран число 102, поскольку она находится в третьей строке программы, отсчитывая от директивы **#line 100**.

```
#include <stdio.h>

#line 100                    /* Устанавливает счетчик строк */
int main(void)               /* Строка 100 */
{                            /* Строка 101 */
  printf("%d\n",__LINE__); /* Строка 102 */

  return 0;
}
```

Директива #pragma

Директива **#pragma** зависит от реализации. Она позволяет передавать компилятору различные команды. Например, компилятор может иметь опцию, предусматривающую трассировку выполнения программы. Эту опцию можно задать с помощью директивы **#pragma**. Детали и допустимые опции перечисляются в документации, сопровождающей компилятор.

Операторы препроцессора # и

Препроцессор имеет два оператора: # и ##. Эти операторы используются в директиве **#define**.

Оператор #, который называется *оператором превращения в строку* (stringize), преобразует свой аргумент в строку, заключенную в кавычки. В качестве примера рассмотрим следующую программу.

```
#include <stdio.h>

#define mkstr(s)  # s

int main(void)
{
  printf(mkstr(Я люблю C++));

  return 0;
}
```

Препроцессор превратит строку

```
printf(mkstr(Я люблю C++));
```

в строку

```
printf("Я люблю C++");
```

Оператор ##, называемый *оператором конкатенации* (pasting), "склеивает" две лексемы.

```
#include <stdio.h>

#define concat(a, b)  a ## b

int main(void)
{
  int xy = 10;
```

```
    printf("%d", concat(x, y));

    return 0;
}
```

Препроцессор преобразует строку

```
printf("%d", concat(x, y));
```

в строку

```
printf("%d", xy);
```

Если эти операторы показались вам странными, вспомните, что в большинстве случаев без них можно спокойно обойтись. В основном они предназначены для обработки некоторых особых ситуаций.

Имена предопределенных макросов

В языке C++ существует шесть встроенных имен макросов:

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus

Первые пять из этих макросов относятся к языку C.

Макросы **__LINE__** и **__FILE__** уже обсуждались в разделе, посвященном директиве **#line**. Кратко говоря, они содержат номер текущей строки и имя компилируемого файла.

Макрос **__DATE__** содержит строку, имеющую вид *месяц/день/год*. В него записывается дата компиляции файла.

Макрос **__TIME__** содержит время компиляции программы, представленное в формате *часы:минуты:секунды*.

Смысл макроса **__STDC__** зависит от реализации. Как правило, если макрос **__STDC__** определен, компилятор строго придерживается стандарта языков C и C++, не допуская никаких нестандартных расширений.

Компилятор, строго поддерживающий стандарт языка C++, определяет макрос **__cplusplus**, значение которого содержит как минимум шесть цифр. Компиляторы, допускающие нестандартные расширения, определяют значение макроса **__cplusplus**, состоящее из пяти и меньшего количества знаков.

Комментарии

Стандарт C89 допускает лишь один вид комментариев, начинающихся символами **/*** и заканчивающихся символами ***/**. Между звездочкой и косой чертой не должно быть пробелов. Компилятор игнорирует текст, заключенный внутри комментариев. Например, следующая программа выведет на экран слово "Привет".

```
#include <stdio.h>
```

```
int main(void)
{
  printf("Привет");
  /* printf("Здесь"); */

  return 0;
}
```

Этот стиль комментариев называется *многострочным* (multiline comment), поскольку такие комментарии могут состоять из нескольких строк.

```
/* Это - пример
многострочного
комментария */
```

Комментарии можно размещать в любом месте программы, с одним исключением: они не могут разрывать ключевые слова или идентификаторы. Например, следующий комментарий вполне корректен.

```
x = 10+ /* Добавляем число */5;
```

В то же время комментарий

```
swi/*это совершенно неправильно*/tch(c) { ...
```

неверен, поскольку ключевое слово не может содержать комментарий. Впрочем, комментарии не следует вставлять в середину выражений, так как это лишь запутывает программу.

Многострочные комментарии не могут быть вложенными. Иначе говоря, один комментарий не может содержать другой. Например, следующий фрагмент кода вызовет ошибку компиляции.

```
/* Это внешний комментарий
  x = y/a;
  /* Это внутренний комментарий - он порождает ошибку */
*/
```

Однострочные комментарии

Язык C++ (и стандарт C99) предусматривает два вида комментариев. Во-первых, многострочные комментарии. Во-вторых, *однострочные комментарии* (single-line comments).

```
// Это - однострочный комментарий.
```

Однострочные комментарии особенно полезны для создания кратких пошаговых описаний программы. Несмотря на то что они не поддерживаются стандартом C89, многие компиляторы языка C позволяют их применять. Кроме того, они были включены в стандарт C99. И последнее: однострочные комментарии можно вкладывать в многострочные.

Комментарии следует размещать там, где необходимы пояснения. Например, все функции, за исключением очень коротких и самоочевидных, должны содержать комментарии, в которых описано их предназначение, способ вызова и возвращаемое значение.

Полный справочник по C++

Часть II

Язык C++

В первой части мы рассмотрели язык C, являющийся подмножеством языка C++. Во второй части описываются свойства, характерные лишь для языка C++. Иными словами, в ней обсуждаются особенности языка C++, которых нет в языке C. Поскольку многие свойства языка C++ предназначены для поддержки объектно-ориентированного про-граммирования (ООП), во второй части также рассматриваются теория и преимущества именно этой парадигмы. Начнем с обзора языка C++.

Полный справочник по

Глава 11

Обзор языка C++

В данной главе содержится обзор основных концепций, воплощенных в языке C++. Объектно-ориентированные свойства языка C++ тесно связаны друг с другом. По этой причине часто бывает трудно описать одно понятие, не затронув других. Чтобы решить эту проблему, мы сначала предлагаем вашему вниманию беглый обзор наиболее важных аспектов языка C++, к которым относятся его история, ключевые свойства, а также различия между традиционной и стандартной версиями языка C++. В остальных главах мы рассмотрим свойства языка C++ более подробно.

Истоки языка C++

Изначально язык C++ представлял собой расширение языка C. Впервые он был разработан в 1979 году Бьярном Страуструпом (Bjarne Stroustrup), сотрудником компании Bell Laboratories в г. Мюррей-Хилл (штат Нью-Джерси). Сначала язык назывался "C с классами", однако в 1983 году его название было изменено на C++.

Хотя язык C был одним из наиболее любимых и широко распространенных профессиональных языков программирования, его мощности уже не хватало для решения основной проблемы: возрастающей сложности программ. Со временем программы становились все более крупными и сложными. Безусловно, язык C — превосходный язык программирования, но его возможности, к сожалению, ограничены. Если размер программы колеблется в пределах от 25000 до 100000 строк кода, она становится настолько сложной, что механизмы языка C не позволяют работать с ней, как с единым целым. Одной из целей языка C++ было преодоление этого препятствия. Язык C++ позволил программистам прекрасно справляться с более крупными и сложными программами.

Большинство усовершенствований, внесенных Страуструпом в язык C, касалось поддержки объектно-ориентированного программирования (ООП). (Краткое описание этой парадигмы программирования содержится в следующем разделе.) Страуструп утверждает, что некоторые объектно-ориентированные свойства языка C++ были унаследованы у другого объектно-ориентированного языка — Simula 67. Таким образом, язык C++ обладает преимуществами двух мощных методов программирования.

С момента изобретения язык C++ модифицировался трижды, каждый раз приобретая все новые свойства. Первый пересмотр его основ состоялся в 1985 году, второй раз он подвергался ревизии в 1990 году. Несколько лет назад началась работа над созданием стандарта языка C++. Для этого была сформирована объединенная комиссия двух организаций по стандартизации: ANSI (American National Standards Institute — Американский институт национальных стандартов) и ISO (International Standards Organization — Организация международных стандартов). Первый черновой вариант стандарта был обнародован 25 января 1994 года. В этом документе комитет ANSI/ISO C++ (членом которого был и сам автор) пытался придерживаться идей, сформулированных Страуструпом, добавив к ним новые возможности. Однако в целом стандарт лишь констатировал состояние языка C++, сложившееся в тот момент.

Вскоре после опубликования первого варианта стандарта произошло событие, оказавшее огромное влияние на развитие языка C++: Александр Степанов (Alexander Stepanov) разработал стандартную библиотеку шаблонов STL (Standard Template Library). Эта библиотека представляет собой набор обобщенных процедур, которые можно применять для обработки данных. Библиотека STL — не только мощный и элегантный, но и очень крупный механизм. Как следствие, члены комитета дружно проголосовали за включение библиотеки STL в стандарт языка C++. Это значительно расширило первоначальные рамки языка C++ но, кроме всего прочего, затормозило процесс стандартизации.

Следует честно признать, что никто не ожидал такой медленной работы комитета по стандартизации. В ходе этой работы в язык C++ было внесено много новых свойств и сделано большое количество мелких усовершенствований. Фактически вер-

сия языка C++, предложенная комитетом, намного сложнее и крупнее, чем первоначальный вариант Страуструпа. Последний черновой вариант стандарта был опубликован 14 ноября 1997 года, и в 1998 году была, наконец, опубликована окончательная версия этого документа. Как правило, эта спецификация языка C++ называется *стандартом языка C++* (Standard C++).

В основу нашей книги положен стандарт языка C++. Кроме того, в нее включены все новые свойства языка C++. Эта версия языка предложена комитетом ANSI/ISO по стандартизации и поддерживается большинством компиляторов.

Что такое объектно-ориентированное программирование

Поскольку стимулом разработки языка C++ являлось объектно-ориентированное программирование (ООП), необходимо понимать его основные принципы. Объектно-ориентированное программирование — довольно мощный механизм. С момента изобретения компьютера методологии программирования резко изменились, в основном из-за возрастающей сложности программ. Например, программы для первых компьютеров представляли собой последовательности машинных команд, записанных в двоичном коде с помощью переключения тумблеров на передней панели машины. Пока длина программ не превышала нескольких сотен строк, этот подход оставался вполне работоспособным. Однако по мере увеличения размера программ возникла необходимость в новом способе программирования, и появился язык ассемблера, позволяющий закодировать машинные инструкции с помощью символов. Размер программы продолжал увеличиваться, поэтому программистам понадобился более мощный язык, позволяющий справляться со сложными алгоритмами. Как известно, первым широко распространенным языком программирования оказался FORTRAN. Несмотря на то что создание этого языка в свое время было огромным шагом вперед, FORTRAN трудно назвать ясным и выразительным языком, позволяющим создавать простые и наглядные программы.

Шестидесятые годы дали толчок развитию структурного программирования. Основными средствами его воплощения стали языки C и Pascal. Структурные языки программирования позволили легко создавать достаточно сложные программы. Кроме того, они характеризуются поддержкой изолированных подпрограмм, локальных переменных, богатым выбором управляющих конструкций и печально известным оператором `goto`. Несмотря на то что структурные языки представляют собой мощное средство, они не позволяют создавать очень крупные проекты.

Итак, каждый новый метод программирования позволял создавать все более сложные и крупные программы, улучшая предыдущие подходы. До изобретения объектно-ориентированного программирования многие проекты достигали границ, за которыми структурный подход уже не работал. Именно для преодоления этих препятствий была создана объектно-ориентированная парадигма.

Объектно-ориентированное программирование унаследовало лучшие идеи структурного программирования и объединило их с новыми понятиями. В результате возник новый способ организации программ. В принципе, есть два способа организации программ: положить в основу ее код (какие действия происходят?) или данные (что изменяется в результате действий?). В рамках структурного подхода главным понятием является код. Иначе говоря, его принцип можно сформулировать так: "код изменяет данные". Так, программа, написанная на структурном языке, например на языке C, определяется ее функциями, каждая из которых может оперировать данными любого типа.

Объектно-ориентированные программы работают иначе. В их основу положены данные, а базовый принцип формулируется так: "данные контролируют доступ

к коду". В объектно-ориентированных языках определяются данные и процедуры, осуществляющие к ним доступ. Итак, тип данных точно определяет, какого рода операции к нему можно применять.

Для поддержки объектно-ориентированного программирования язык должен обладать тремя свойствами: инкапсуляцией, полиморфизмом и наследованием. Рассмотрим каждое из этих понятий отдельно.

Инкапсуляция

Инкапсуляция (incapsulation) — это механизм, связывающий воедино код и данные, которыми он манипулирует, а также обеспечивающий их защиту от внешнего вмешательства и неправильного использования. В объектно-ориентированном языке код и данные можно погружать в "черный ящик", который называется *объектом* (object). Иначе говоря, объект — это средство инкапсуляции.

Внутри объекта код и данные могут быть *закрытыми* (private) или *открытыми* (public). Закрытый код или данные объекта доступны только из другой части этого же объекта. Иначе говоря, к закрытой части кода или данных невозможно обратиться извне. Если код или данные являются открытыми, они доступны из любой части программы. Как правило, открытая часть кода обеспечивает управляемое взаимодействие (интерфейс) с закрытыми элементами объекта.

Как с синтаксической, так и с семантической точки зрения объект представляет собой переменную, тип которой определен пользователем. Людям, привыкшим к структурному программированию, может показаться странным, что переменная может содержать не только данные, но и код. Однако в объектно-ориентированном программировании дело обстоит именно так. Определяя новый тип объекта, вы определяете новый тип данных. Каждый экземпляр этого типа данных является сложной переменной.

Полиморфизм

Языки объектно-ориентированного программирования поддерживают *полиморфизм* (polymorphism), который характеризуется фразой "один интерфейс, несколько методов". Проще говоря, полиморфизм — это атрибут, позволяющий с помощью одного интерфейса управлять доступом к целому классу методов. Конкретный выбор определяется возникшей ситуацией. В реальном мире примером полиморфизма является термостат. Независимо от вида топлива (газ, мазут, электричество и т.д.), термостат работает одинаково. В данном случае термостат (представляющий собой интерфейс) остается неизменным, независимо от типа печки (метода). Например, если вы хотите поднять температуру до 70 градусов, нужно просто установить термостат на соответствующее деление, независимо от типа печки.

Этот принцип распространяется и на программирование. Например, допустим, что в программе определены три разных типа стека. Один стек состоит из целых чисел, другой — из символов, а третий — из чисел с плавающей точкой. Благодаря полиморфизму программисту достаточно определить функции `push()` и `pop()`, которые можно применять к любому типу стека. В программе необходимо создать три разные версии этих функций для каждой разновидности стека, но имена этих функций должны быть одинаковыми. Компилятор автоматически выберет правильную функцию, основываясь на информации о типе данных, хранящихся в стеке. Таким образом, функции `push()` и `pop()` — интерфейс стека — одинаковы, независимо от типа стека. Конкретные варианты этих функций определяют конкретные реализации (методы) для каждого типа данных.

Полиморфизм позволяет упростить программу, создавая один интерфейс для выполнения разных действий. Ответственность за выбор *конкретного действия* (метода) в возникшей ситуации перекладывается на компилятор. Программисту не обязательно вмешиваться в этот процесс. Нужно лишь помнить правила и правильно применять *общий интерфейс*.

Первые объектно-ориентированные языки были интерпретируемыми, поэтому полиморфизм был динамическим. Однако язык C++ является компилируемым. Таким образом, в языке C++ поддерживается как динамический, так и статический полиморфизм.

Наследование

Наследование (inheritance) — это процесс, в ходе которого один объект может приобретать свойства другого. Он имеет большое значение, поскольку поддерживает концепцию *классификации* (classification). Если подумать, все знания организованы по принципу иерархической классификации. Например, антоновка относится в классу "*яблоки*", который в свою очередь является частью класса "*фрукты*", входящего в класс "*пища*". Если бы классификации не существовало, мы не смогли бы точно описать свойства объектов. Однако при этом необходимо указывать только уникальные свойства объекта, позволяющие выделить его среди других объектов данного класса. Именно этот принцип лежит в основе механизма наследования, дающего возможность считать конкретный объект специфическим экземпляром более общей разновидности. В дальнейшем мы убедимся, что наследование является важным аспектом объектно-ориентированного программирования.

Некоторые основные принципы языка C++

В первой части книги было описано подмножество С языка C++, и все примеры представляли собой программы, фактически написанные на языке С. Начиная с этого момента, все примеры будут программами, написанными на языке C++, т.е. будут иллюстрировать свойства, присущие лишь языку C++. Для краткости назлвем эти характеристики "свойствами C++".

Программисты, имеющие опыт работы на языке С, должны помнить, что C++ существенно отличается от языка С. Большинство этих отличий относится к объектно-ориентированным свойствам языка. Однако программы, написанные на C++, отличаются также способом ввода-вывода данных и особенностями работы с заголовками. Кроме того, большинство программ на языке C++ обладают специфическими свойствами, характерными именно для языка C++. Перед изучением объектно-ориентированных конструкций языка C++ следует тщательно изучить основные элементы программ, написанных на нем.

В этом разделе рассматриваются вопросы, касающиеся практически всех программ на языке C++. По ходу изложения мы также будем подчеркивать различия между стандартом языка C++, языком С и ранними версиями языка C++.

Пример программы на языке C++

Рассмотрим простую программу на языке C++.

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  cout << "Это — вывод.\n";  // Однострочный комментарий
  /* Комментарий в стиле языка С */

  // Ввод числа с помощью оператора >>
  cout << "Введите число: ";
  cin >> i;

  // Теперь выведем число с помощью оператора <<
```

```
    cout << i << " в квадрате равно " << i*i << "\n";

    return 0;
}
```

Очевидно, что эта программа сильно отличается от программ, приведенных в первой части. В ней есть весьма полезный однострочный комментарий. Она начинается с включения заголовка **<iostream>**, поддерживающего ввод-вывод в стиле языка C++. (Аналогом заголовка **<iostream>** в языке C является заголовочный файл **<stdio.h>**.) Обратите внимание на еще одну особенность: заголовок **iostream** не имеет расширения **.h**, поскольку стандарт языка C++ не использует старый стиль.

Следующая строка программы

```
using namespace std;
```

сообщает компилятору, что следует использовать пространство имен **std**. Пространство имен представляет собой новшество, недавно включенное в язык C++. Пространство имен создает область видимости, в которой могут размещаться различные элементы программы. Они позволяют организовывать большие программы. Оператор **using** информирует компилятор о том, что программист желает использовать пространство имен **std**. В этом пространстве имен объявлена вся стандартная библиотека языка C++. Таким образом, пространство имен **std** упрощает доступ к стандартной библиотеке. Программы, включенные в первую часть книги, использовали только язык C. Пространства имен в них были не нужны, поскольку по умолчанию библиотечные функции языка C находятся в глобальном пространстве имен.

На заметку *Поскольку заголовки в новом стиле и пространства имен включены в язык C++ лишь недавно, старые программы их не используют. Кроме того, старые компиляторы также не поддерживают эти механизмы. Как поступать в таких ситуациях, мы расскажем ниже.*

Рассмотрим теперь следующую строку.

```
int main()
```

Обратите внимание на то, что список параметров функции **main()** пуст. Этим она отличается от языка C, в котором функция, не имеющая параметров, должна вместо списка параметров использовать ключевое слово **void**.

```
int main(void)
```

Именно так объявлялась функция **main()** в первой части книги. Однако в языке C++ использовать для этого ключевое слово **void** совершенно не обязательно. Как правило, в языке C++, если функция не имеет параметров, список параметров просто не указывается и слово **void** в скобках записывать не нужно.

Следующая строка иллюстрирует два свойства языка C++.

```
cout << "Это — вывод.\n";  // Однострочный комментарий
```

Во-первых, оператор

```
cout << "Это — вывод.\n";
```

выводит на экран сообщение "**Это — вывод.**" и выполняет переход на следующую строку. В языке C++ оператор "**<<**" имеет несколько значений. Он по-прежнему является оператором побитового сдвига влево, но, как показывает данный пример, также используется в качестве *оператора вывода*. Слово **cout** представляет собой идентификатор, связанный с экраном. (На самом деле язык C++, как и язык C, поддерживает перенаправление ввода-вывода, но для простоты будем предполагать, что идентифика-

тор **cout** всегда связан с экраном.) Идентификатор **cout** и оператор "<<" можно применять для вывода данных любого встроенного типа, а также символьных строк.

Обратите внимание на то, что для вывода можно по-прежнему использовать функцию **printf()** и другие функции, предусмотренные для ввода-вывода данных в языке С. Однако большинство программистов полагают, что использование оператора "<<" больше соответствует духу языка С++. Более того, хотя применение функции **printf()** при выводе строк фактически эквивалентно оператору "<<", система ввода-вывода языка С++ предоставляет программистам более широкие возможности, позволяя вводить и выводить объекты, а не только данные встроенных типов, как в языке С.

Вслед за оператором вывода в программе помещен *однострочный комментарий* (single-line comment). Как указывалось в главе 10, в языке С++ есть два вида комментариев. Во-первых, в программах можно по-прежнему использовать многострочные комментарии в стиле языка С. Кроме них, используя символы **//**, можно определять однострочные комментарии. Компилятор игнорирует их содержимое комментариев. Как правило, программисты используют многострочные комментарии для пространных описаний, а однострочные — для кратких замечаний.

Затем программа предлагает пользователю ввести число, которое считывается с клавиатуры с помощью оператора

```
cin >> i;
```

В языке С++ оператор ">>" по-прежнему является оператором побитового сдвига вправо. Однако, как показано выше, в языке С++ он играет роль *оператора вывода*. Этот оператор присваивает переменной **i** значение, введенное с клавиатуры. Идентификатор **cin** означает стандартный поток ввода, обычно связанный с клавиатурой. Как правило, выражение **cin** ">>" позволяет вводить данные любых встроенных типов, а также строки.

На заметку

Описанная выше строка кода не содержит опечаток. В частности, перед идентификатором **i** *не следует ставить символ* **&**. *При вводе информации с помощью функций языка С, например, функции* **scanf()**, *необходимо передавать указатель на переменную, получающую значение. Именно по этой причине перед ее именем в языке С необходимо указывать оператор получения адреса* **&**. *Однако оператор ">>" в языке С++ реализован иначе, поэтому он не использует (точнее говоря, не должен использовать) оператор получения адреса* **&**. *Причины объясняются в главе 13.*

В программе на языке С++ вместо оператора ">>" можно применять любые функции ввода, предусмотренные языком С, в частности, функцию **scanf()**, хотя в данном примере эта возможность не использовалась. Однако, как и при использовании оператора "<<", большинство программистов полагают, что оператор ">>" больше соответствует духу языка С++.

Рассмотрим еще одну интересную строку программы.

```
cout << i << "в квадрате равно " << i*i << "\n";
```

Предположим, что переменная **i** имеет значение 10. Тогда данный оператор выведет на экран фразу "**10 в квадрате равно 100**". Как видим, в одной и той же строке программы можно использовать несколько операторов << подряд.

Программа завершается оператором

```
return 0;
```

Он возвращает вызывающему процессу (как правило, операционной системе) нулевое значение. В программе на языке С++ этот оператор работает точно так же, как и в программе на языке С. Нулевое значение является признаком нормального завершения работы программы. Аварийное завершение должно сопровождаться возвращением ненулевого значения. Кроме того, в этом случае можно применять именованные константы **EXIT_SUCCESS** и **EXIT_FAILURE**.

Операторы ввода-вывода

Операторы "<<" и ">>" можно применять соответственно для вывода и ввода данных любого встроенного типа. Например, следующая программа вводит данные типа **float**, **double** и строку, а затем выводит их на экран.

```
#include <iostream>
using namespace std;

int main()
{
  float f;
  char str[80];
  double d;

  cout << "введите два числа с плавающей запятой: ";
  cin >> f >> d;

  cout << "Введите строку: ";
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

Запустите эту программу и попробуйте ввести строку "**Это проверка**", когда она предложит это сделать. Программа выведет на экран лишь слово "**Это**". Остальная часть строки будет проигнорирована, поскольку за словом "**Это**" следует разделитель. Таким образом, строка "проверка" не считывается программой. Кроме того, этот пример показывает, что в одной и той же строке можно применять несколько операторов ввода "<<" подряд.

Операторы ввода-вывода, предусмотренные в языке C++, реагируют на все эскейп-последовательности, перечисленные в главе 2. Например, следующий фрагмент является совершенно правильным.

```
cout << "A\tB\tC";
```

Этот оператор выводит на экран символы A, B и C, разделенные табуляцией.

Объявление локальных переменных

Если вы знаете язык C, обратите внимание на важное различие между языками C и C++, касающееся объявления локальных переменных. В стандарте C89 локальные переменные каждого блока должны быть объявлены в его начале. Переменные нельзя объявлять после выполняемого оператора. Например, в стандарте C89 следующий пример является неправильным.

```
/* Фрагмент, неправильный в языке C.
   В языке C++ это вполне допускается */
int f()
{
  int i;
  i = 10;

  int j;  /* В программах на языке C так делать нельзя. */
  j = i*2;

  return j;
}
```

В программах, соответствующих стандарту C89, эта функция вызовет сообщение об ошибке, поскольку между объявлениями переменных **i** и **j** находится оператор присваивания. Однако, если скомпилировать этот фрагмент в программе на языке C++, все будет в порядке. В языке C++ (и C99) переменные можно объявлять в любом месте программы, а не только в начале блоков.

Рассмотрим еще один пример. Это вариант программы из предыдущего раздела, в котором переменная **str** объявляется только тогда, когда в ней возникает необходимость.

```
#include <iostream>
using namespace std;

int main()
{
  float f;
  double d;
  cout << "Enter two floating point numbers: ";
  cin >> f >> d;

  cout << "введите строку: ";
  char str[80];  // Строка str объявляется здесь.
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

Программист может сам решать, где ему удобнее объявлять переменные. Однако, следуя принципам инкапсуляции кода и данных, переменные рекомендуется объявлять поближе к месту их применения. В предыдущем примере объявления переменных были разделены просто для демонстрации такой возможности, но в других ситуациях это свойство языка может оказаться намного ценнее.

Объявление переменных непосредственно перед их использованием позволяет избежать возможных побочных эффектов. Однако наибольшую пользу это приносит в больших функциях. Честно говоря, в коротких функциях (к которым относится большинство наших примеров), нет особого смысла распределять объявления переменных по всему тексту. Достаточно просто объявить их в самом начале функции. Поэтому мы будем применять этот прием сообразно размерам и сложности создаваемых функций.

У специалистов нет общего мнения по поводу локализации объявления переменных. Некоторые из них считают, что объявления, разбросанные по всему тексту, затрудняют анализ программ. По этой причине программисты не придают этой возможности большого значения. В нашей книге мы не принимаем чью-либо сторону. И все же в особых случаях, в частности в больших функциях, объявление переменных непосредственно перед их использованием облегчает отладку программ.

Правило “int по умолчанию”

Несколько лет назад в языке C++ появилось новшество, которое может повлиять на совместимость новых программ и кода, написанного на старых версиях C++ и на языке C. В стандарте C89 и в исходном варианте языка C++ отсутствие явного типа в объявлении переменной по умолчанию трактовалось как **int**. Однако правило “**int** по умолчанию” при стандартизации было отброшено. В стандарте C99 его также нет. И все же на практике по-прежнему часто встречаются программы на языке C и старые программы на языке C++, в которых используется это правило.

В большинстве случаев это правило относится к типу значения, возвращаемого функцией. В частности, стало общепринятой практикой не указывать тип возвращае-

мого значения **int**, если функция возвращает целое число. Например, стандарт C89 и старые версии языка C++ вполне допускают следующий фрагмент программы.

```
func(int i)
{
  return i*i;
}
```

Стандарт языка C++ требует явно указывать тип возвращаемого значения, даже если возвращается целое число.

```
int func(int i)
{
  return i*i;
}
```

По практическим соображениям большинство компиляторов по-прежнему придерживаются правила "**int** по умолчанию". Однако при создании новых программ это правило следует забыть.

Тип данных bool

В языке C++ определен встроенный булев тип с именем **bool**. Объекты типа **bool** могут принимать только два значения: **true** и **false**. Как указывалось в первой части книги, булевы переменные автоматически преобразуются в целые числа, и наоборот. В частности, все ненулевые значения преобразуются в значение **true**, а нуль — в значение **false**. Справедливо и обратное утверждение: значение **true** преобразуется в единицу, а **false** — в нуль. Таким образом, в языке C++ сохраняется принцип, согласно которому нуль считается ложным значением, а любое ненулевое значение — истинным.

На заметку

*Хотя стандарт C89 (подмножество языка C++) не предусматривает булев тип, в стандарте C99 был добавлен тип **_Bool**, позволяющий хранить значения 1 и 0 (т.е. истинное и ложное). В отличие от языка C++ в стандарте C99 нет ключевых слов **true** и **false**. Следовательно, тип **_Bool**, определенный в стандарте C99, несовместим с типом **bool**, определенным в языке C++.*

*Дело в том, что во многих существующих программах, написанных на языке C, программисты определяли свой собственный вариант типа **bool**. Чтобы сохранить совместимость с этими программами, в стандарт C99 включили неполноценный тип **_Bool**. Однако совместимости между стандартом C99 и C++ достичь все же можно, поскольку в стандарте C99 существует заголовочный файл **<stdbool.h>**, в котором определены макросы **bool**, **true** и **false**. Включив в программу этот заголовочный файл, можно достичь совместимости между стандартом C99 и языком C++.*

Старый и новый стиль языка C++

В процессе своего развития и стандартизации язык C++ подвергся значительным изменениям. В результате фактически существуют две версии языка C++. Первая из них — традиционная, разработанная Бьярном Страуструпом. Вторая версия была разработана Страуструпом и комитетом ANSI/ISO по стандартизации. Они очень похожи друг на друга, хотя вторая версия обладает большими возможностями. Таким образом, традиционный язык C++ является подмножеством стандартного.

В нашей книге описывается стандартный вариант языка C++. Эта версия разработана комитетом ANSI/ISO по стандартизации и поддерживается всеми современными компиляторами. Программы, приведенные в книге, написаны в современном стиле и строго следуют стандарту. Однако, если вы работаете со старым компилятором, не-

которые программы могут создать проблемы. В процессе стандартизации комитет ANSI/ISO внес в язык значительные усовершенствования. Стремясь соответствовать стандарту, разработчики компиляторов учли их в своих программах. Разумеется, между появлением новой возможности языка и ее реализацией в коммерческих компиляторах всегда проходит определенное время. Поскольку новшества вносились в язык на протяжении нескольких лет, старые версии компиляторов не смогли их учесть. Это очень важно, поскольку два последних усовершенствования касаются буквально всех программ, включая самые простые. Если ваш компилятор не поддерживает эти возможности, не печальтесь — эти препятствия легко обойти.

Принципиальная разница между старым и новым стилями сводится к двум свойствам: новый стиль заголовков и использование оператора **namespace**. Чтобы понять эти различия, рассмотрим две простейшие программы на языке C++. Первая из них соответствует старому стилю программирования.

```
/*
   Программа на C++ в старом стиле.
*/

#include <iostream.h>

int main()
{
  return 0;
}
```

Обратите особое внимание на директиву **#include**. Она включает в текст программы заголовочный файл **iostream.h**, а не заголовок **<iostream>**. Кроме того, в этом варианте программы отсутствует оператор **namespace**.

Рассмотрим вторую скелетную программу, соответствующую новому стилю.

```
/*
   Современный стиль программы на языке C++,
   использующий новый вид заголовков и пространство имен.
*/
#include <iostream>
using namespace std;

int main()
{
  return 0;
}
```

В этом варианте программы используется новый вид заголовков и пространство имен. Оба эти свойства мы уже упоминали. Рассмотрим их поближе.

Новый стиль заголовков

Как известно, для вызова стандартной библиотечной функции программа должна содержать заголовочный файл. Для этого в программу включают директиву **#include**. Например, в языке C, чтобы ввести в программу заголовочный файл **<stdio.h>**, необходимо выполнить следующий оператор.

```
#include <stdio.h>
```

Здесь параметр **stdio.h** представляет собой имя файла, в котором объявлены прототипы функций ввода-вывода, а директива препроцессора **#include** вынуждает компилятор включить этот файл в текст исходной программы. Итак, следует помнить, что директива **#include** *включает файл* в текст программы.

На протяжении нескольких лет после своего появления язык C++ использовал тот же стиль работы с заголовками, что и язык C. Иными словами, в нем применялись *заголовочные файлы*. Стандартный C++ также допускает использование заголовочных файлов в стиле языка C, позволяя программистам сохранять обратную совместимость программ. Однако стандарт языка C++ предусматривает новый вид заголовка, который используется в стандартной библиотеке. Эти заголовки *не являются файлами*. Они просто представляют собой некие стандартные идентификаторы, которые могут соответствовать файлам, а могут не соответствовать им. Новые заголовки являются абстракцией, гарантирующей, что в программе будут объявлены соответствующие прототипы и определения, необходимые для вызова библиотечных функций.

Поскольку заголовки нового стиля не являются именами файлов, у них нет расширения **.h**. Они состоят лишь из имени заголовка, заключенного в угловые скобки. Рассмотрим, например, некоторые стандартные заголовки нового стиля.

```
<iostream>   <fstream>   <vector>   <string>
```

Новые заголовки включаются в программу с помощью директивы **#include**. Единственная разница состоит в том, что заголовки нового стиля не обязательно являются именами файлов.

Поскольку язык C++ унаследовал все библиотеки функций языка C, он продолжает поддерживать стандартные заголовочные файлы языка C. Иначе говоря, заголовочные файлы, такие как **stdio.h** или **ctype.h**, по-прежнему доступны. Однако стандарт языка C++, кроме них, определяет заголовки нового вида, которые могут заменять собою реальные заголовочные файлы. В программах на языке C++ к именам заголовочных файлов языка C добавляется префикс "**c**", а расширение "**.h**" отбрасывается. Например, заголовочному файлу **math.h** в старом стиле соответствует заголовок **<cmath>**, а файлу **string.h** — заголовок **<cstring>**. Несмотря на то что использование заголовочных файлов старого образца допускается, в стандарте языка C++ этого делать не рекомендуется. В нашей книге мы придерживаемся нового стиля заголовков. Если ваш компилятор не поддерживает новый стиль заголовков, можно просто заменить их именами заголовочных файлов старого образца.

Поскольку новый стиль заголовков появился относительно недавно, многие старые программы не используют его. В них по-прежнему применяются заголовочные файлы в стиле языка C. Например, имя заголовочного файла, содержащего прототипы функций ввода-вывода, указывается так.

```
#include <iostream.h>
```

Эта директива вынуждает компилятор включить в программу заголовочный файл **iostream.h**. Как правило, имена заголовков старого и нового стиля совпадают, за исключением расширения "**.h**".

Как известно, все компиляторы допускают использование заголовочных файлов старого стиля. Однако этот стиль считается устаревшим, и его не рекомендуется применять в новых программах.

Внимание! *Несмотря на то что заголовки старого стиля все еще применяются в существующих программах на C++, они считаются устаревшими.*

Пространства имен

При включении в программу заголовка нового стиля его содержимое погружается в пространство имен **std**. *Пространство имен* — это просто область видимости. Оно предназначено для локализации имен идентификаторов и предотвращения конфликтов между ними. Элементы, объявленные в одном пространстве имен, отделены от элементов, принадлежащих другому пространству. Изначально имена библиотечных

функций просто размещались в глобальном пространстве имен (как в языке С). Однако после появления заголовков нового образца их содержимое стали размещать в пространстве имен **std**. Позднее мы рассмотрим этот вопрос более подробно, а пока эта проблема не должна нас волновать, так как оператор

```
using namespace std;
```

погружает пространство имен в область видимости (т.е. пространство имен **std** включается в глобальное пространство имен). Этот оператор уничтожает разницу между заголовками старого и нового стиля.

И еще одно: чтобы сохранить совместимость с языком С, принято следующее соглашение. Если программа, написанная на языке С++, содержит заголовок старого стиля, его содержимое погружается в глобальное пространство имен. Это позволяет компилятору языка С++ компилировать программы, написанные на языке С.

Работа со старым компилятором

Как мы уже говорили, пространства имен и заголовки нового стиля появились относительно недавно. Они были добавлены в процессе стандартизации языка С++. Несмотря на то что все новые компиляторы языка С++ поддерживают эти свойства, старые компиляторы могут их не учитывать. В этом случае компилятор выдает на экран сообщения об ошибках. Эту проблему очень легко решить: нужно лишь заменить заголовки именами заголовочных файлов старого стиля и удалить оператор **namespace**. Иначе говоря, следует заменить операторы

```
#include <iostream>
using namespace std;
```

оператором

```
#include <iostream.h>
```

Это изменение превращает современную программу в старомодную. Поскольку содержимое всех заголовочных файлов погружается в глобальное пространство имен, оператор **namespace** теперь не нужен.

И последнее: в обозримом будущем на практике еще долго будут встречаться программы, написанные в старом стиле, использующие заголовочные файлы и не применяющие оператор **using**. Все современные компиляторы справляются с ними без проблем. Однако новые программы следует писать в современном стиле, поскольку лишь он поддерживается стандартом языка С++. Несмотря на то что старомодные программы еще многие годы будут поддерживаться компиляторами, следует помнить, что их структура является слишком жесткой.

Введение в классы

В этом разделе рассматривается самая важная часть языка С++ — класс. Для того чтобы создать объект, в языке С++ сначала необходимо определить его общий вид, используя ключевое слово **class**. С синтаксической точки зрения класс похож на структуру. Рассмотрим в качестве примера класс, определяющий тип под названием **stack**. С его помощью можно создать реальный стек.

```
#define SIZE 100

// В этом фрагменте объявляется класс stack.
class stack {
  int stck[SIZE];
```

```
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};
```

Класс может содержать открытую и закрытую части. По умолчанию все члены класса считаются закрытыми. Например, переменные **stck** и **tos** являются закрытыми. Это означает, что к ним невозможно обратиться из функций, не являющихся членами класса **stack**. Таким образом достигается инкапсуляция — доступ к закрытым элементам класса строго контролируется. В классе можно также определить закрытые функции, которые могут вызываться другими членами класса.

Для того чтобы открыть элементы класса (т.е. сделать их доступными для других частей программы), следует объявить их с помощью ключевого слова **public**. Все переменные или функции, размещенные в разделе **public**, доступны для любых функций программы. По существу, внешняя часть программы получает доступ к объекту именно через его открытые функции-члены. Хотя переменные можно объявлять открытыми, этого следует избегать. Наоборот, все данные рекомендуется объявлять закрытыми, контролируя доступ к ним с помощью открытых функций. Кроме того, обратите внимание на то, что после ключевого слова **public** в определении класса **stack** стоит двоеточие.

Функции **init()**, **push()** и **pop()** называются *функциями-членами*, поскольку они являются частью класса **stack**. Переменные **stck** и **tos** называются *переменными-членами* (или *данными-членами*). Напомним, объект создает связь между кодом и данными. Только функции-члены имеют доступ к закрытым членам своего класса. Следовательно, доступ к переменным **stck** и **tos** имеют только функции **init()**, **push()** и **pop()**.

Определив класс, можно создать объект этого типа. В сущности, имя класса становится новым спецификатором типа данных. Например, следующий оператор создает объект с именем **mystack** типа **stack**.

```
stack mystack;
```

Объявляя объект класса, вы создаете *экземпляр* (instance) этого класса. В данном случае переменная **mystack** является экземпляром класса **stack**. Кроме того, объекты можно создавать, указывая их имя сразу после определения класса, т.е. после закрывающей фигурной скобки (точно так же, как экземпляр структуры).

В языке C++ с помощью ключевого слова **class** определяется новый тип данных, который можно использовать для создания объектов этого типа. Следовательно, объект — это экземпляр класса. В этом смысле он ничем не отличается от других переменных, например, от переменной, представляющей собой экземпляр типа **int**. Иначе говоря, класс является логической абстракцией, а объект — ее реальным воплощением, существующим в памяти компьютера.

Определение простого класса имеет следующий вид.

```
class имя_класса
{
    закрытые переменные и функции
public:
    открытые переменные и функции
}  список имен объектов;
```

Естественно, *список имен объектов* может быть пустым.

Внутри определения класса **stack** функции-члены идентифицируются с помощью своих прототипов. Напомним, что в языке C++ все функции должны иметь прототипы. Таким образом, прототипы являются неотъемлемой частью класса. Прототип функции-члена, размещенный внутри определения класса, имеет тот же смысл, что и прототип обычной функции.

Кодируя функцию-член класса, необходимо сообщить компьютеру, какому именно классу она принадлежит. Для этого перед ее именем следует указать имя соответствующего класса. Вот как, например, определяется функция **push()** из класса **stack**.

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Стек переполнен.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}
```

Оператор "**::**" называется *оператором разрешения области видимости* (scope resolution operator). Он сообщает компилятору, что данная версия функции **push()** принадлежит классу **stack**, т.е. функция **push()** пребывает в области видимости класса **stack**. Благодаря оператору разрешения области видимости компилятор всегда знает, какому классу принадлежит каждая из функция.

Ссылаясь на член класса извне, всегда следует указывать конкретный объект, которому этот член принадлежит. Для этого используется имя объекта, за которым следует оператор "**.**" и имя члена. Это правило относится как к данным-членам, так и к функциям-членам. Вот как, например, вызывается функция **init()**, принадлежащая объекту **stack1**.

```
stack stack1, stack2;

stack1.init();
```

В этом фрагменте создаются два объекта — **stack1** и **stack2**, а затем объект **stack1** инициализируется с помощью функции **init()**. Обратите внимание на то, что переменные **stack1** и **stack2** представляют собой два разных объекта. Таким образом, инициализация объекта **stack1** *не означает* инициализации объекта **stack2**. Объекты **stack1** и **stack2** связывает лишь то, что они являются экземплярами одного и того же класса.

Внутри класса функции-члены могут вызывать друг друга и обращаться в переменным-членам, не используя оператор ".". Этот оператор необходим, лишь когда обращение в функциям и переменным-членам осуществляется извне класса.

Рассмотрим завершенную программу, в которой определяется класс **stack**.

```
#include <iostream>
using namespace std;

#define SIZE 100

// Определение класса stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};

void stack::init()
{
  tos = 0;
}
```

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Стек полон.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Стек пуст.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack stack1, stack2;  // Создаем два объекта класса stack

  stack1.init();
  stack2.init();

  stack1.push(1);
  stack2.push(2);

  stack1.push(3);
  stack2.push(4);

  cout << stack1.pop() << " ";
  cout << stack1.pop() << " ";
  cout << stack2.pop() << " ";
  cout << stack2.pop() << "\n";

  return 0;
}
```

Результат работы этой программы показан ниже.

```
3  1  4  2
```

Не забывайте: закрытые члены объекта доступны только функциям-членам, принадлежащим этому же объекту. Например, в предыдущем примере оператор

```
stack1.tos = 0; // Ошибка, переменная tos является закрытой.
```

нельзя было бы поместить в функцию **main()**, поскольку переменная **tos** является закрытым членом объекта **stack1.**

Перегрузка функций

Перегрузка функций является одной из разновидностей полиморфизма. В языке C++ допускается использование функций с одинаковыми именами, но разными объявлениями параметров. В таких случаях говорят, что функции *перегружены*, а сам процесс называется *перегрузкой функций*.

Чтобы понять, насколько важен механизм перегрузки, рассмотрим три функции, определенные в подмножестве C: **abs()**, **labs()** и **fabs()**. Функция **abs()** возвращает абсолютное значение целого числа, функция **labs()** вычисляет абсолютное значение переменной типа **long int**, а функция **fabs()** предназначена для определения абсолютного значения переменной типа **double**. Несмотря на то что эти функции выполняют практически идентичные операции, в языке C для них предусмотрено три разных, хотя и мало отличающихся, имени. Это слишком усложняет ситуацию, программист должен постоянно помнить, какую функцию следует вызывать в том или ином случае. Однако в языке C++ все три функции можно назвать одинаково. Рассмотрим пример.

```
#include <iostream>
using namespace std;

// Функция abs перегружена трижды.
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
  cout << abs(-10) << "\n";

  cout << abs(-11.0) << "\n";

  cout << abs(-9L) << "\n";

  return 0;
}

int abs(int i)
{
  cout << "Функция abs() с аргументом типа int\n";

  return i<0 ? -i : i;
}

double abs(double d)
{
  cout << "Функция abs() с аргументом типа double\n";

  return d<0.0 ? -d : d;
}

long abs(long l)
{
  cout << "Функция abs() с аргументом типа long\n";

  return l<0 ? -l : l;
}
```

Эта программа вычисляет следующие результаты.

```
Функция abs с аргументом типа int:
10
Функция abs с аргументом типа double:
11
Функция abs с аргументом типа long:
9
```

В этой программе определены три похожие, но разные функции с именем **abs()**, каждая из которых возвращает абсолютное значение своего аргумента. В каждом конкретном случае компилятор автоматически выбирает подходящую функцию, основываясь на информации о типе аргумента. Ценность механизма перегрузки заключается в том, что он позволяет обращаться к функциям, близким по смыслу, используя одно и то же имя. Таким образом, имя **abs()** является названием *общей операции*, а компилятор сам должен выбрать ее *конкретную реализацию* в зависимости от контекста. Нужно лишь помнить, какая общая операция должна быть выполнена. Итак, благодаря полиморфизму три сущности были заменены одной. Разумеется, этот пример слишком тривиален, но, развивая эту концепцию, можно применить полиморфизм для управления работой более сложных программ.

Как правило, чтобы перегрузить какую-то функцию, достаточно определить три ее разные версии, а компилятор сам позаботится об остальном. Однако следует помнить об одном важном ограничении: тип и/или количество параметров каждой перегруженной функции должны быть разными. Нельзя перегрузить функции, отличающиеся лишь типом возвращаемого значения. Этого совершенно недостаточно, поскольку необходимо, чтобы типы и/или количество параметров в каждой версии отличались друг от друга. (Информации о типе возвращаемого значения недостаточно для того, чтобы компилятор правильно распознал нужный вариант перегруженной функции.) Разумеется, это не значит, что перегружаемые функции *должны* иметь одинаковый тип возвращаемого значения. Просто для перегрузки этого мало.

Рассмотрим еще один пример перегруженных функций.

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

void stradd(char *s1, char *s2);
void stradd(char *s1, int i);

int main()
{
  char str[80];

  strcpy(str, "Всем ");
  stradd(str, "привет");
  cout << str << "\n";

  stradd(str, 100);
  cout << str << "\n";

  return 0;
}

// Конкатенация двух строк
void stradd(char *s1, char *s2)
{
  strcat(s1, s2);
}

// Конкатенация строки и целого числа, преобразованного в строку
void stradd(char *s1, int i)
{
  char temp[80];

  sprintf(temp, "%d", i);
  strcat(s1, temp);
}
```

В этой программе функция **stradd()** является перегруженной. Одна ее версия склеивает две строки (аналогично функции **strcat()**). Другая версия сначала преобразует целое число в строку, а затем конкатенирует полученные строки. Таким образом, благодаря перегрузке можно создать единый интерфейс, позволяющий конкатенировать две строки, а также строку и целое число.

Разумеется, одно и то же имя можно использовать и для функций, совершенно не связанных между собой, но делать этого не следует. Например, именем **sqr()** можно назвать функцию, вычисляющую *квадрат* целого числа и извлекающую *квадратный корень* из числа с плавающей точкой. Однако эти операции принципиально разные, поэтому их перегрузка не имеет никакого смысла (это считается дурным тоном). Перегрузка нужна лишь для создания общего интерфейса для родственных, тесно связанных между собой операций.

Перегрузка операторов

Полиморфизм в языке C++ проявляется также в виде перегрузки операторов. Как известно, для ввода и вывода в языке C++ применяются операторы "**>>**" и "**<<**". Это стало возможным благодаря тому, что в заголовке **<iostream>** эти операторы перегружены. Перегруженные операторы имеют разный смысл для разных классов, сохраняя при этом свое первоначальное значение.

В языке C++ можно перегрузить почти все операторы. Вернемся, например, к классу **stack**, рассмотренному нами выше. В этом классе можно перегрузить оператор "**+**", заставив его складывать два объекта класса **stack**. При этом оператор "**+**" сохраняет свой первичный смысл.

Поскольку механизм перегрузки операторов намного сложнее, чем перегрузка функций, отложим его обсуждение до главы 14.

Наследование

Как указывалось ранее, наследование является одной из основных характерных черт объектно-ориентированного программирования. В языке C++ наследование выражается в том, что один класс может приобретать свойства другого класса в момент своего объявления. Наследование позволяет создавать иерархии классов, уточняя их свойства от самых общих до более конкретных. Процесс наследования начинается с определения *базового класса* (base class), свойства которого будут общими для всех его наследников. Базовый класс представляет собой описание наиболее общего характера. Наследники базового класса называются *производными классами* (derived classes). Производный класс обладает всеми свойствами базового класса и своими специфическими особенностями. Продемонстрируем этот механизм на примере, в котором описываются разные типы зданий.

Сначала объявляется класс **building**. Он служит основой для создания двух производных классов.

```
class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
```

```
  void set_area(int num);
  int get_area();
};
```

Для простоты все здания описываются лишь тремя свойствами: количеством комнат, количеством этажей и общей площадью. Класс **building** включает эти компоненты в своем объявлении. Функции-члены, имена которых начинаются с префикса **set**, задают значения закрытых данных. Функции-члены, имена которых начинаются с префикса **get**, извлекают и возвращают эти значения.

Теперь применим это слишком общее описание здания для объявления производных классов, описывающих более конкретные виды сооружений. Вот как, например, выглядит класс **house**.

```
// Класс house является производным от класса building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};
```

Обратите внимание на то, как происходит наследование свойств класса **building**. В общем случае производный класс объявляется следующим образом.

```
class производный_класс:спецификатор_доступа базовый_класс
{
   // тело нового класса
}
```

Здесь *спецификатор_доступа* указывать необязательно. Этот параметр может принимать три значения: **public**, **private** или **protected**. (Подробнее их смысл описан в главе 12.) Пока будем считать, что все производные классы используют открытое наследование. Ключевое слово **public** означает, что все открытые члены базового класса становятся открытыми членами производного класса. Следовательно, открытые члены класса **building** становятся открытыми членами производного класса **house** и доступны функциям-членам класса **house**, как будто они объявлены непосредственно внутри производного класса. Однако функции-члены класса **house** *не имеют* доступа в закрытым членам класса **building**. Это очень важный момент. Несмотря на то что класс **house** является наследником класса **building**, он имеет доступ лишь к открытым членам класса **building**. Таким образом, наследование не позволяет нарушить принципы инкапсуляции информации, положенные в основу объектно-ориентированного программирования.

Внимание! *Производный класс имеет прямой доступ как к своим членам, так и к открытым членам базового класса.*

Рассмотрим программу, демонстрирующую механизм наследования. Она создает два класса, производных от класса **building**: классы **house** и **school**.

```
#include <iostream>
using namespace std;

class building {
  int rooms;
  int floors;
  int area;
```

```
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};

// Класс house является производным от класса building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};

// Класс school также является производным от класса building
class school : public building {
  int classrooms;
  int offices;
public:
  void set_classrooms(int num);
  int get_classrooms();
  void set_offices(int num);
  int get_offices();
};

void building::set_rooms(int num)
{
  rooms = num;
}

void building::set_floors(int num)
{
  floors = num;
}

void building::set_area(int num)
{
  area = num;
}

int building::get_rooms()
{
  return rooms;
}

int building::get_floors()
{
  return floors;
}

int building::get_area()
{
  return area;
}
```

```
void house::set_bedrooms(int num)
{
  bedrooms = num;
}

void house::set_baths(int num)
{
  baths = num;
}

int house::get_bedrooms()
{
  return bedrooms;
}

int house::get_baths()
{
  return baths;
}

void school::set_classrooms(int num)
{
  classrooms = num;
}

void school::set_offices(int num)
{
  offices = num;
}

int school::get_classrooms()
{
  return classrooms;
}

int school::get_offices()
{
  return offices;
}

int main()
{
  house h;
  school s;

  h.set_rooms(12);
  h.set_floors(3);
  h.set_area(4500);
  h.set_bedrooms(5);
  h.set_baths(3);

  cout << "В доме " << h.get_bedrooms();
  cout << " спален\n";

  s.set_rooms(200);
  s.set_classrooms(180);
  s.set_offices(5);
  s.set_area(25000);
```

```
  cout << "В школе " << s.get_classrooms();
  cout << " классных комнат\n";
  cout << "Ее площадь равна " << s.get_area();

  return 0;
}
```

Результат работы этой программы приведен ниже.

```
В доме 5 спален
В школе 180 классных комнат
Ее площадь равна 25000
```

Как видим, основное преимущество механизма наследования заключается в том, что можно сначала создать общую классификацию, а затем на ее основе разработать уточняющие классы. Таким образом, каждый объект может точно представлять свой собственный подкласс.

В книгах о языке C++ отношение наследования часто описывают с помощью терминов *базовый класс* и *производный класс*. Однако наряду с ними часто упоминаются термины *предок* и *наследник*. Кроме того, можно встретить понятия *суперкласс* и *подкласс*.

Кроме возможности создавать иерархическую классификацию, наследование обеспечивает поддержку динамического полиморфизма с помощью механизма виртуальных функций. (Детали изложены в главе 16.)

Конструкторы и деструкторы

Очень часто некоторая часть объекта перед его первым использованием должна быть инициализирована. Например, вернемся к описанию класса **stack**. Перед первым использованием объекта этого класса переменной **tos** следует присвоить число 0. Для этого предназначена функция **init()**. Поскольку инициализация объектов — очень распространенное требование, в языке C++ предусмотрен особый механизм, позволяющий инициализировать объекты в момент их создания. Эта автоматическая инициализация осуществляется конструктором.

Конструктор — это особая функция, являющаяся членом класса. Ее имя должно совпадать с именем класса. Например, класс **stack** можно модифицировать, предусмотрев в нем конструктор.

```
// Класс stack с конструктором.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // Конструктор
  void push(int i);
  int pop();
};
```

Обратите внимание на то, что в объявлении конструктора **stack()** не указан тип возвращаемого значения, поскольку в языке C++ конструкторы в принципе не могут возвращать значения.

Код конструктора **stack()** может выглядеть так.

```
// Конструктор класса stack
stack::stack()
{
```

```
  tos = 0;
  cout << "Стек инициализирован\n";
}
```

Учтите, что сообщение "**Стек инициализирован**" просто иллюстрирует работу конструктора. На практике конструкторы чаще всего ничего не вводят и не выводят: они просто выполняют различные виды инициализации.

Конструктор автоматически вызывается в момент создания объекта, т.е. при его объявлении. Следовательно, объявление объекта в языке C++ — это не пассивная запись, а активный процесс. В языке C++ объявление является выполняемым оператором. Это различие носит вовсе не академический характер. Код, с помощью которого конструируется объект, может быть довольно важным. Для глобальных и статических локальных объектов конструкторы вызываются лишь однажды. При объявлении локальных объектов конструкторы вызываются каждый раз при входе в соответствующий блок.

Антиподом конструктора является *деструктор*. Во многих ситуациях объект должен выполнить некоторое действие или действия, которые уничтожат его. Локальные объекты создаются при входе в соответствующий блок, а при выходе из него они уничтожаются. При разрушении объекта автоматически вызывается его деструктор. Для этого существует несколько причин. Например, объект должен освободить занимаемую им память или закрыть файл, открытый им ранее. В языке C++ эти действия выполняет деструктор. Имя деструктора должно совпадать с именем конструктора, но перед ним ставится знак ~ (тильда). Рассмотрим класс **stack**, содержащий конструктор и деструктор. (Учтите, что на самом деле класс **stack** не нуждается в деструкторе, здесь он приведен в качестве иллюстрации.)

```
// Объявление класса stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // Конструктор
  ~stack(); // Деструктор
  void push(int i);
  int pop();
};

// Конструктор класса stack
stack::stack()
{
  tos = 0;
  cout << "Стек инициализирован\n";
}

// Деструктор класса stack
stack::~stack()
{
  cout << "Стек уничтожен\n";
}
```

Учтите, что конструкторы и деструкторы не возвращают никаких значений.

Проиллюстрируем работу конструктора и деструктора с помощью новой версии программы **stack**. Обратите внимание на то, что теперь функция **init()** не нужна.

```
// Использование конструктора и деструктора.
#include <iostream>
using namespace std;

#define SIZE 100
```

```
// Создаем класс stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // Конструктор
  ~stack(); // Деструктор
  void push(int i);
  int pop();
};

// Конструктор класса stack
stack::stack()
{
  tos = 0;
  cout << "Стек инициализирован\n";
}

// Деструктор класса stack
stack::~stack()
{
  cout << "Стек уничтожен\n";
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Стек полон.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Стек пуст.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack a, b;  // Создаем два объекта класса stack

  a.push(1);
  b.push(2);

  a.push(3);
  b.push(4);

  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";
```

```
  return 0;
}
```

Результат работы программы выглядит так.

```
Стек инициализирован
Стек инициализирован
3 1 4 2
Стек уничтожен
Стек уничтожен
```

Ключевые слова языка C++

Стандарт языка C++ содержит 63 ключевых слова. Они приведены в табл. 11.1.

Таблица 11.1. Ключевые слова языка C++

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Структура программы на языке C++

Программа на языке C++ имеет такой общий вид.

```
#директивы include
объявления базовых классов
объявления производных классов
прототипы обычных функций
int main()
{
 // ...
}
определения обычных функций
```

В большинстве крупных проектов все объявления классов должны размещаться в отдельном заголовочном файле, который может включаться в каждый модуль. Однако общая структура программы остается неизменной.

В оставшихся главах мы более подробно рассмотрим описанные выше свойства, а также другие аспекты языка C++.

Полный справочник по

Глава 12

Классы и объекты

В основе объектно-ориентированного программирования на языке C++ лежит понятие класса. Класс определяет природу объекта и является основным механизмом инкапсуляции. В этой главе мы рассмотрим классы и объекты более подробно.

Классы

Классы создаются с помощью ключевого слова **class**. Объявление класса определяет новый тип, связывающий код и данные между собой. Таким образом, класс является логической абстракцией, а объект — ее физическим воплощением. Иными словами, объект — это экземпляр класса.

Объявление класса очень похоже на объявление структуры. В главе 11 был продемонстрирован упрощенный вид такого объявления. Ниже приводится полная общая форма объявления класса, который не является наследником никакого другого класса.

```
class имя-класса {
      закрытые данные и функции
спецификатор_доступа:
      данные и функции
спецификатор_доступа:
      данные и функции
   // ...
спецификатор_доступа:
      данные и функции
} список_объектов;
```

Список объектов указывать не обязательно. Он просто позволяет объявлять объекты класса. В качестве *спефицикатора доступа* используется одно из трех ключевых слов языка C++:

```
public
private
protected
```

По умолчанию функции и данные, объявленные внутри класса, считаются закрытыми и доступны лишь функциям — членам этого класса. Спецификатор **public** открывает доступ к функциям и данным класса из других частей программы. Спецификатор доступа **protected** необходим только при наследовании классов (см. главу 15). Зона влияния спецификатора доступа простирается до следующего спецификатора или до конца объявления файла.

Внутри объявления класса спецификаторы доступа можно чередовать в произвольном порядке. Например, некоторые члены класса можно сделать открытыми, а затем вновь перейти к объявлению закрытых переменных и функций. Проиллюстрируем сказанное следующим примером.

```
#include <iostream>
#include <cstring>
using namespace std;

class employee {
  char name[80]; // Закрыт по умолчанию
public:
  void putname(char *n); // Открытые члены
  void getname(char *n);
private:
  double wage; // Снова объявляются закрытые члены
```

```
public:
  void putwage(double w); // Вновь объявляются открытые члены
  double getwage();
};

void employee::putname(char *n)
{
  strcpy(name, n);
}

void employee::getname(char *n)
{
  strcpy(n, name);
}

void employee::putwage(double w)
{
  wage = w;
}

double employee::getwage()
{
  return wage;
}

int main()
{
  employee ted;
  char name[80];

  ted.putname("Тед Джонс");
  ted.putwage(75000);

  ted.getname(name);
  cout << name << " зарабатывает $";
  cout << ted.getwage() << " за год.";

  return 0;
}
```

Здесь класс **employee** используется для хранения имени сотрудника и величины его оклада. Обратите внимание на то, что спецификатор доступа **public** использован дважды.

Несмотря на то что ограничений на использование спецификаторов доступа нет, этой возможностью не стоит злоупотреблять. Она полезна лишь для визуализации разных групп переменных, связанных между собой, и улучшает наглядность программы. Однако компилятору все равно, как сгруппированы данные внутри объявления класса. Многие программисты считают более естественным группировать все открытые, защищенные и закрытые данные, используя не более одного раздела каждого вида. Например, большинство программистов записало бы объявление класса **employee** так, как показано ниже.

```
class employee {
  char name[80];
  double wage;
public:
  void putname(char *n);
  void getname(char *n);
  void putwage(double w);
  double getwage();
};
```

Функции, объявленные внутри класса, называются *функциями-членами*. Они имеют доступ ко всем элементам своего класса, включая закрытые члены. Переменные, объявленные внутри класса, называются *переменными-членами* или *данными-членами*. (Используется также термин *экземпляр переменной*.) Итак, любой элемент класса называется *членом класса*.

Существует несколько ограничений на применение членов класса. Нестатические члены класса не могут иметь инициализатор. Объявление класса не может содержать объявление своего объекта. (Хотя членом класса может быть указатель на объект данного класса.) Члены класса не могут объявляться с ключевыми словами **auto**, **extern** и **register**.

Как правило, все данные-члены объявляют закрытыми. Это обеспечивает инкапсуляцию данных. Однако существуют ситуации, в которых некоторые переменные следует объявлять открытыми. (Например, если некоторая переменная используется очень часто, открытый доступ выполняется намного быстрее.) Синтаксическая конструкция для доступа к открытому члену класса ничем не отличается от вызова функции-члена: указывается имя объекта, оператор "." и имя переменной. Рассмотрим простой пример.

```
#include <iostream>
using namespace std;

class myclass {
public:
  int i, j, k; // Доступны из любой точки программы
};

int main()
{
  myclass a, b;

  a.i = 100; // Переменные i, j и k доступны
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12; // Напомним, что переменные a.k и b.k
            // принадлежат разным объектам
  cout << a.k << " " << b.k;

  return 0;
}
```

Связь между структурами и классами

Структуры являются наследием языка C. Как мы уже знаем, синтаксические конструкции класса и структуры очень похожи. Однако связь между структурой и классом намного теснее, чем кажется на первый взгляд. В языке C++ роль структуры была значительно расширена. Фактически она стала альтернативой классу. Единственное различие между структурой и классом заключается в том, что все члены структуры по умолчанию считаются открытыми, а все члены класса — закрытыми. Во всех других отношениях структуры и классы эквивалентны. Иначе говоря, *в языке C++ структура является разновидностью класса*. Рассмотрим пример программы, в которой структура используется в качестве класса, контролирующего доступ к строке.

```
// Использование структуры в качестве класса.
#include <iostream>
#include <cstring>
```

```
using namespace std;

struct mystr {
  void buildstr(char *s); // Открытый член
  void showstr();
private: // Закрытый член
  char str[255];
} ;

void mystr::buildstr(char *s)
{
  if(!*s) *str = '\0'; // Инициализация строки
  else strcat(str, s);
}

void mystr::showstr()
{
  cout << str << "\n";
}

int main()
{
  mystr s;

  s.buildstr(""); // Инициализация
  s.buildstr("Всем ");
  s.buildstr("привет!");

  s.showstr();

  return 0;
}
```

Эта программа выводит на экран строку "**Всем привет!**".

Класс **mystr** можно переписать с помощью ключевого слова **class**.

```
class mystr {
  char str[255];
public:
  void buildstr(char *s); // Открытый член
  void showstr();
} ;
```

Возникает естественный вопрос: "Зачем в языке C++ предусмотрены два эквивалентных ключевых слова **struct** и **class**?". Эта мнимая избыточность объясняется несколькими причинами. Во-первых, не нужно ограничивать возможности структур. В языке C они уже использовались для группировки данных. Следовательно, можно сделать небольшой шаг вперед и включить в структуры функции-члены. Во-вторых, поскольку структуры и классы тесно связаны между собой, программы, написанные на языке C, легко трансформировать в программы на языке C++. И, наконец, несмотря на то что структуры и классы практически эквивалентны, применение двух альтернатив сохраняет свободу модификации для ключевого слова **class**, поскольку ключевое слово **struct** навсегда связано с языком C и обеспечивает обратную совместимость программ.

Хотя ключевое слово **struct** может заменять слово **class**, большинство программистов предпочитают этого не делать. Обычно ключевое слово **class** следует применять для объявления класса, а ключевое слово **struct** — для объявления структуры в стиле C. Именно такой стиль принят в нашей книге. Иногда для описания структуры

в стиле C применяется аббревиатура *POD* (Plain Old Data — простые старые данные). Это значит, что такая структура не содержит конструкторов, деструктора и функций-членов.

Внимание! *В языке C++ объявление структуры определяет тип класса.*

Связь между объединениями и классами

Кроме структур, для определения класса используются объединения **union**. В языке C++ объединения могут содержать не только данные, но и функции. Они могут включать конструкторы и деструктор. Объединения сохраняют все свойства, предусмотренные языком C, в частности, их члены могут размещаться в одной и той же области памяти. Как и структуры, члены объединения по умолчанию считаются открытыми и полностью совместимы с языком C. В приведенном ниже примере объединение используется для перестановки байтов в переменной типа **unsigned short**. (Предполагается, что целое значение занимает 2 байт.)

```
#include <iostream>
using namespace std;

union swap_byte {
  void swap();
  void set_byte(unsigned short i);
  void show_word();

  unsigned short u;
  unsigned char c[2];
};

void swap_byte::swap()
{
  unsigned char t;

  t = c[0];
  c[0] = c[1];
  c[1] = t;
}

void swap_byte::show_word()
{
  cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
  u = i;
}

int main()
{
  swap_byte b;

  b.set_byte(49034);
  b.swap();
  b.show_word();
```

```
  return 0;
}
```

Объявление объединения в языке C++ определяет особый вид класса. Это значит, что принцип инкапсуляции не нарушается.

Существует несколько ограничений, наложенных на применение объединений в языке C++. Во-первых, объединения не могут использовать механизм наследования. Во-вторых, объединение не может служить базовым классом. Объединение не может содержать: 1) виртуальные функции; 2) статические переменные и ссылки; 3) объекты классов, в которых перегружен оператор присваивания; и, наконец, 4) объекты классов, в которых явно заданы конструкторы или деструктор.

К объединениям, не содержащим функции-члены, конструкторы и деструктор, также можно применять термин POD.

Безымянные объединения

В языке C++ есть особая разновидность объединений — безымянные объединения, которые не имеют типа и не могут образовывать объекты. Вместо этого безымянное объединение сообщает компилятору, что его члены хранятся в одной области памяти. Однако доступ к этим переменным осуществляется непосредственно, без помощи оператора ".". Рассмотрим следующую программу.

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  // Определение безымянного объединения
  union {
    long l;
    double d;
    char s[4];
  } ;

  // Теперь открыт прямой доступ к элементам объединения.
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

Как видим, элементы объединения ничем не отличаются от обычных локальных переменных. Несмотря на то что они объявлены внутри объединения, их область видимости распространяется на весь блок. Отсюда следует, что имена членов безымянных объединений не должны конфликтовать с другими идентификаторами в той же области видимости.

Все ограничения, наложенные на обычные объединения, распространяются и на безымянные. Во-первых, в них могут содержаться лишь данные (функции-члены не допускаются). Во-вторых, безымянные объединения не могут содержать закрытые и защищенные элементы. В заключение, глобальные безымянные объединения должны объявляться с помощью ключевого слова **static**.

Дружественные функции

С помощью ключевого слова **friend** можно предоставить обычной функции доступ к закрытым членам класса. Дружественная функция имеет доступ ко всем закрытым и защищенным членам класса. Для того чтобы объявить дружественную функцию, следует поместить ее прототип внутри класса, указав перед ней ключевое слово **friend**. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Внимание: функция sum() не является функцией-членом
// никакого класса.
int sum(myclass x)
{
  /* Так как функция sum() является дружественной по
     отношению к классу myclass, она имеет прямой доступ
     к переменным a и b. */

  return x.a + x.b;
}

int main()
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

В данном примере функция **sum()** не является членом класса **myclass**. Однако она обладает полным доступом к закрытым членам. Кроме того, функцию **sum()** можно вызывать без помощи оператора ".". Поскольку она не является членом класса, ей не нужно указывать имя объекта.

Хотя дружественные функции имеют равные права с членами класса, в некоторых ситуациях они бывают особенно полезны. Во-первых, дружественные функции позволяют перегружать некоторые виды операторов (см. главу 14). Во-вторых, они облегчают создание некоторых функций ввода-вывода (см. главу 18). В-третьих, дружественные функции полезны в ситуациях, когда несколько классов могут содержать члены, тесно связанные с другими частями программы. Рассмотрим последнее утверждение подробнее.

Представьте себе два класса, каждый из которых выводит на экран сообщение об ошибке. Другая часть программы должна проверять, отображается ли в данный момент на экране сообщение от ошибке, чтобы не повредить его при выводе своей информации. Эту проблему можно решить, поместив в каждый класс функцию-член, возвращающую индикатор активного сообщения. Однако для проверки этого условия придется выполнить дополнительную работу, т.е. вызвать две функции, а не одну. Если условие нужно проверить быстро, дополнительные затраты времени могут оказаться неприемлемыми. Однако, если объявить функцию, дружественную по отношению к каждому из классов, можно проверить состояние обоих объектов с помощью одной функции. В таких ситуациях дружественные функции позволяют создавать более эффективный код. Проиллюстрируем это утверждение следующей программой.

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // неполное объявление

class C1 {
  int status;  // Равно IDLE, если экран свободен,
               // и INUSE, если экран занят.
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
  int status; // Равно IDLE, если экран занят,
              // и INUSE, если экран свободен.
  //
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}

int main()
{
  C1 x;
  C2 y;
```

```
  x.set_status(IDLE);
  y.set_status(IDLE);

  if(idle(x, y)) cout << "Экран занят.\n";
  else cout << "In use.\n";

  x.set_status(INUSE);

  if(idle(x, y)) cout << "Экран свободен.\n";
  else cout << "Экран занят.\n";

  return 0;
}
```

Обратите внимание на то, что в этой программе используется *неполное объявление* (forward declaration) класса **C2**. Это необходимо, поскольку объявление функции **idle()** в классе **C1** ссылается на класс **C2**, который еще не объявлен. Чтобы осуществить неполное объявление класса, можно просто применить способ, продемонстрированный в программе.

Дружественная функция может быть членом другого класса. Рассмотрим вариант нашей программы, включив функцию **idle()** в класс **C1**.

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2;  // Неполное объявление

class C1 {
  int status;  // Равно IDLE, если экран занят,
               // и INUSE, если свободен.
  // ...
public:
  void set_status(int state);
  int idle(C2 b);  // Теперь функция idle() — член класса C1.
};

class C2 {
  int status;  // Равно IDLE, если экран занят,
               // и INUSE, если свободен.
  // ...
public:
  void set_status(int state);
  friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}
```

```
// Функция idle() является членом класса C1 и другом класса C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main()
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Экран свободен.\n";
  else cout << "Экран занят.\n";
  x.set_status(INUSE);

  if(x.idle(y)) cout << "Экран свободен.\n";
  else cout << "Экран занят.\n";

  return 0;
}
```

Поскольку функция **idle()** является членом класса **C1**, она имеет прямой доступ к переменной **status**, принадлежащей объекту класса **C1**. Таким образом, любой объект класса **C2** необходимо передавать в функцию **idle()**.

Дружественные функции имеют два важных ограничения. Во-первых, производный класс не наследует дружественные функции. Во-вторых, дружественные функции не могут содержать спецификатор хранения, т.е. в ее объявлении нельзя использовать ключевые слова **static** или **extern**.

Дружественные классы

Один класс может быть дружественным по отношению к другому. В этом случае дружественный класс и все его функции-члены имеют доступ к закрытым членам, определенным в другом классе. Рассмотрим пример.

```
// Применение дружественных классов.
#include <iostream>
using namespace std;

class TwoValues {
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min {
public:
  int min(TwoValues x);
};
```

```
int Min::min(TwoValues x)
{
  return x.a < x.b ? x.a : x.b;
}

int main()
{
  TwoValues ob(10, 20);
  Min m;

  cout << m.min(ob);

  return 0;
}
```

В данном примере класс **Min** имеет доступ к закрытым переменным **a** и **b**, объявленным в классе **TwoValues**.

Следует запомнить: если один класс является дружественным по отношению к другому, он просто получает доступ к сущностям, определенным в этом классе, но не наследует их, т.е. члены класса не становятся членами дружественного класса.

Дружественные классы редко используются в практических приложениях. Они необходимы лишь в особых случаях.

Подставляемые функции

Язык C++ обладает важным свойством: в нем существуют *подставляемые функции* (inline functions), которые широко используются в классах. В оставшейся части главы (и на протяжении всей книги) мы будем часто применять подставляемые функции, поэтому рассмотрим их подробнее.

В языке C++ можно написать короткую функцию, которая не вызывается, а подставляется в соответствующее место программы. Этот процесс напоминает функциональную макроподстановку. Чтобы заменить вызов функции подстановкой, перед ее определением следует указать слово **inline**. Например, в следующей программе функция **max()** не вызывается, а подставляется.

```
#include <iostream>
using namespace std;

inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main()
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

С точки зрения компилятора эта программа выглядит так.

```
#include <iostream>
using namespace std;
```

```
int main()
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```

Подставляемые функции позволяют создавать очень эффективные программы. Поскольку классы обычно содержат несколько интерфейсных функций, которые зачастую вызываются для доступа к его закрытым членам, необходимо, чтобы эти функции выполнялись как можно быстрее. Как известно, каждый вызов функции сопряжен с дополнительными затратами на передачу и возврат управления. Обычно при вызове функции ее аргументы заталкиваются в стек, а содержимое регистров копируется в оперативную память, чтобы после возврата управления можно было восстановить первоначальное состояние программы. На эти операции затрачивается дополнительное время. Однако, если вместо вызова тело функции просто подставляется в программу, ничего этого не требуется. К сожалению, ускорение работы программы достигается за счет увеличения размера кода, поскольку тело подставляемой функции дублируется несколько раз. По этой причине подставляемые функции должны быть очень маленькими. Кроме того, подставляемыми следует делать только те функции, быстродействие которых действительно существенно влияет на эффективность программы.

Как и спецификатор **register**, ключевое слово **inline** является лишь *рекомендацией*, а не приказом компилятору. В некоторых случаях компилятор может его проигнорировать. Кроме того, некоторые компиляторы ограничивают категории функций, которые могут быть подставляемыми. В частности, как правило, компиляторы не разрешают подставлять рекурсивные функции. В каждом конкретном случае информацию об ограничениях на применение подставляемых функций следует искать в документации, сопровождающей компилятор. Учтите, если функцию нельзя подставить, она будет вызываться.

Подставляемые функции могут быть членами класса. Например, следующая программа считается вполне корректной.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void init(int i, int j);
  void show();
};

// Создать подставляемую функцию.
inline void myclass::init(int i, int j)
{
  a = i;
  b = j;
}

// Создать другую подставляемую функцию.
inline void myclass::show()
{
  cout << a << " " << b << "\n";
}
```

```
int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

На заметку *Ключевое слово* ***inline*** *не является частью языка С. Следовательно, оно не определено в стандарте С89, но включено в стандарт С99.*

Определение подставляемых функций внутри класса

Короткую функцию можно определить непосредственно внутри объявления класса. Если функция определена внутри объявления класса, она автоматически превращается в подставляемую (если это не противоречит ограничениям компилятора). Указывать при этом ключевое слово **inline** совершенно не обязательно, хотя ошибкой это не считается. Например, предыдущую программу можно переписать, поместив определения функций **init()** и **show()** в объявление класса **myclass**.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // Автоматическая подстановка
  void init(int i, int j) { a=i; b=j; }
  void show() { cout << a << " " << b << "\n"; }
};

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

Обратите внимание на способ записи тела функции внутри объявления класса **myclass**. Поскольку подставляемые функции обычно короткие, такой прием вполне типичен. Однако совершенно необязательно записывать функции именно так. Например, объявление функции **myclass** можно переписать иначе.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // Автоматическая подстановка
  void init(int i, int j)
```

```
  {
    a = i;
    b = j;
  }

  void show()
  {
    cout << a << " " << b << "\n";
  }
};
```

С технической точки зрения подстановка функции **show()** не имеет смысла, поскольку время, затрачиваемое на ввод-вывод, намного превышает время, необходимое для вызова функции. Однако подавляющее большинство программистов предпочитает помещать все короткие функции-члены внутри объявления класса. (Крайне редко в профессиональных программах можно встретить короткие функции-члены, определенные вне объявления класса.)

Конструктор и деструктор могут быть подставляемыми либо по умолчанию, если они определены внутри объявления класса, либо явно.

Конструкторы с параметрами

Конструкторам можно передавать аргументы, предназначенные для инициализации объекта. Параметры конструктора задаются так же, как и для любой другой функции. Рассмотрим класс, конструктор которого имеет параметры.

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  myclass(int i, int j) {a=i; b=j;}
  void show() {cout << a << " " << b;}
};

int main()
{
  myclass ob(3, 5);

  ob.show();

  return 0;
}
```

Обратите внимание на то, что в определении конструктора **myclass()** параметры **i** и **j** используются для инициализации членов **a** и **b**.

Эта программа иллюстрирует наиболее распространенный способ задания аргументов при объявлении объекта, использующего конструктор с параметрами. Например, оператор

```
myclass ob(3, 4);
```

создает объект **ob** и передает аргументам **i** и **j** конструктора **myclass()** значения 3 и 4. Существует еще один способ передачи аргументов:

```
myclass ob = myclass(3, 4);
```

Однако первый способ используется гораздо чаще. На самом деле разница между этими двумя объявлениями невелика. Она связана с конструктором копирования, который мы обсудим в главе 14.

Рассмотрим пример, иллюстрирующий применение конструктора с параметрами. В нем объявлен класс, объекты которого хранят информацию о библиотечных книгах.

```
#include <iostream>
#include <cstring>
using namespace std;

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[40];
  char title[40];
  int status;
public:
  book(char *n, char *t, int s);
  int get_status() {return status;}
  void set_status(int s) {status = s;}
  void show();
};

book::book(char *n, char *t, int s)
{
  strcpy(author, n);
  strcpy(title, t);
  status = s;
}

void book::show()
{
  cout << title << "." << author;
  cout << " находится ";
  if(status==IN) cout << "в библиотеке.\n";
  else cout << "у читателя.\n";
}

int main()
{
  book b1("Твен", "Том Сойер", IN);
  book b2("Мелвилл", "Моби Дик", CHECKED_OUT);

  b1.show();
  b2.show();

  return 0;
}
```

Конструкторы с параметрами позволяют избежать применения отдельных функций для инициализации одной или нескольких переменных в объекте. Чем меньше функций вызывается в программе, тем выше ее эффективность. Кроме того, обратите внимание на то, что короткие функции **get_status()** и **set_status()** определены внутри объявления класса **book**, т.е. сделаны подставляемыми. Этот прием часто применяется при создании программ на языке C++.

Конструкторы с одним параметром: особый случай

Если конструктор имеет один параметр, возникает третий способ передачи начального значения. Рассмотрим следующий пример.

```
#include <iostream>
using namespace std;

class X {
  int a;
public:
  X(int j) { a = j; }
  int geta() { return a; }
};

int main()
{
  X ob = 99; // Передает параметру j значение 99.

  cout << ob.geta(); // Выводит на экран число 99.

  return 0;
}
```

Здесь конструктор класса **X** имеет один параметр. Обратите особое внимание на способ объявления объекта **ob** в функции **main()**. В этом случае число 99 автоматически передается параметру **j** конструктора **X()**. Иначе говоря, компилятор считает этот оператор эквивалентным следующему:

```
X ob = X(99);
```

Как правило, если конструктор имеет один аргумент, объект можно инициализировать либо с помощью выражения *ob(i)*, либо с помощью оператора *ob=j*. Это позволяет неявно выполнять преобразование типа аргумента в тип класса.

Помните, что эта альтернатива относится лишь к конструкторам, имеющим только один параметр.

Статические члены класса

Члены класса (как функции, так и переменные) могут быть статическими. Рассмотрим эту тему подробнее.

Статические переменные-члены

Если перед объявлением переменной-члена поставить ключевое слово **static**, компилятор создаст только один экземпляр этой переменной, который будет использоваться всеми объектами данного класса. В отличие от обычных переменных-членов, статические переменные-члены не копируются для каждого объекта отдельно. Независимо от количества объектов класса, статическая переменная всегда существует в одном экземпляре. Таким образом, все объекты данного класса используют одну и ту же переменную. Все статические переменные инициализируются нулем еще до создания первого объекта класса.

Объявление статической переменной-члена в классе не означает ее *определения* (иначе говоря, память для нее не выделяется). Чтобы разместить статическую пере-

менную в памяти, следует определить ее вне класса, т.е. глобально. Для этого в повторном объявлении статической переменной перед ее именем указывается имя класса, которому она принадлежит, и оператор разрешения области видимости. (Напомним, что объявление класса представляет собой всего лишь логическую схему, а не физическую сущность.)

Чтобы разобраться в механизме использования статических переменных-членов, рассмотрим следующую программу.

```
#include <iostream>
using namespace std;

class shared {
  static int a;
  int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show();
} ;

int shared::a; // Определяем переменную a.

void shared::show()
{
  cout << "Это статическая переменная a: " << a;
  cout << "\nЭто обычная переменная b: " << b;
  cout << "\n";
}

int main()
{
  shared x, y;

  x.set(1, 1); // Присваиваем переменной a значение 1
  x.show();

  y.set(2, 2); // Присваиваем переменной b значение 2
  y.show();

  x.show(); /* Здесь одновременно изменяются значения
               переменных-членов объектов x и y, поскольку
               переменная a используется обоими объектами. */

  return 0;
}
```

Результат работы этой программы таков.

```
Это статическая переменная a: 1
Это обычная переменная b: 1
Это статическая переменная a: 2
Это обычная переменная b: 2
Это статическая переменная a: 1
Это обычная переменная a: 1
```

Обратите внимание на то, что целочисленная переменная **a** объявлена как внутри класса **shared**, так и вне его. Мы уже указывали, что это необходимо, поскольку объявление переменной **a** внутри класса **shared** не сопровождается выделением памяти.

На заметку

Старые версии языка C++ не требовали вторичного объявления статической переменной-члена, что приводило к серьезным недоразумениям, и через несколько лет от этого соглашения отказались. Однако на практике еще встречаются старые программы, которые не содержат вторичного определения статических переменных-членов. В этих случаях следует внести в программу необходимые исправления.

Статическая переменная-член возникает *до* создания первого объекта класса. Например, в следующей короткой программе переменная **a** одновременно является и открытой, и статической. Следовательно, функция **main()** имеет к ней прямой доступ. Поскольку переменная **a** возникает раньше объектов класса **share**, ей можно присвоить значение в самом начале программы. Как показано в программе, при создании объекта **x** значение переменной **a** не изменяется. По этой причине оба оператора вывода отображают на экране одно и то же значение — 99.

```
#include <iostream>
using namespace std;

class shared {
public:
  static int a;
} ;

int shared::a; // Определение переменной a.

int main()
{
  // Инициализация перед созданием объекта.
  shared::a = 99;

  cout << "Начальное значение переменной a: " << shared::a;
  cout << "\n";

  shared x;

  cout << "Значение переменной x.a: " << x.a;

  return 0;
}
```

Обратите внимание на то, что при обращении к переменной **a** необходимо указывать имя класса, которому она принадлежит, и применять оператор разрешения области видимости. Как правило, чтобы обратиться к статической переменной-члену независимо от объекта, необходимо всегда указывать имя класса, в котором она объявлена.

Статические переменные-члены позволяют управлять доступом к ресурсам, которые совместно используются всеми объектами класса. Например, можно создать несколько объектов, записывающих данные на жесткий диск. Однако очевидно, что в каждый конкретный момент времени запись может производить лишь один объект. В таких ситуациях следует объявить статическую переменную, служащую индикатором. По значению этой переменной можно определить, свободен файл или занят. Каждый из объектов должен анализировать это значение перед началом записи. Рассмотрим пример, иллюстрирующий эту ситуацию.

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
```

```
  int get_resource();
  void free_resource() {resource = 0;}
};

int c1::resource; // Определяем ресурс.

int c1::get_resource()
{
  if(resource) return 0; // Ресурс занят.
  else {
    resource = 1;
    return 1;  // Ресурс предоставлен объекту.
  }
}

int main()
{
  c1 ob1, ob2;

  if(ob1.get_resource()) cout << "Объект ob1 обладает ресурсом\n";

  if(!ob2.get_resource()) cout << "Объекту ob2 доступ запрещен\n";

  ob1.free_resource();  // Освобождаем ресурс.

  if(ob2.get_resource())
    cout << "Объект ob2 может использовать ресурс\n";

  return 0;
}
```

С помощью статической переменной можно также определить количество существующих объектов конкретного класса. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class Counter {
public:
  static int count;
  Counter() { count++; }
  ~Counter() { count--; }
};
int Counter::count;

void f();

int main(void)
{
  Counter o1;
  cout << "Существующие объекты: ";
  cout << Counter::count << "\n";

  Counter o2;
  cout << "Существующие объекты: ";
  cout << Counter::count << "\n";

  f();
  cout << "Существующие объекты: ";
  cout << Counter::count << "\n";
```

```
  return 0;
}

void f()
{
  Counter temp;
  cout << "Существующие объекты: ";
  cout << Counter::count << "\n";
  // После возврата из функции f() переменная temp уничтожается
}
```

Эта программа выводит на экран следующие строки.

```
Существующие объекты: 1
Существующие объекты: 2
Существующие объекты: 3
Существующие объекты: 2
```

Как видим, статическая переменная-член **count** увеличивается при каждом создании объекта и уменьшается при каждом уничтожении. Следовательно, с ее помощью можно отследить количество существующих объектов класса **Counter**.

Статические переменные-члены не заменяют собой глобальные. Однако глобальные переменные не соответствуют духу объектно-ориентированного программирования, поскольку они нарушают принципы инкапсуляции.

Статические функции-члены

Функции-члены также могут быть статическими, но на них распространяется несколько ограничений. Они имеют прямой доступ только к другим статическим членам класса. (Разумеется, глобальные функции и данные также доступны статическим функциям-членам.) Статическая функция-член не имеет указателя **this**. (Этот указатель описывается в главе 13.) Одна и та же функция не может одновременно иметь статическую и нестатическую версии. Статические функции не могут быть виртуальными. И в заключение, статические функции нельзя объявлять с помощью ключевых слов **const** или **volatile**.

Рассмотрим слегка измененную версию программы, предоставляющей один и тот же ресурс разным объектам. Обратите внимание на то, что функцию **get_resource()** можно вызывать как независимо от объекта, используя имя класса и оператор разрешения области видимости, так и через объект.

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  static int get_resource();
  void free_resource() { resource = 0; }
};

int cl::resource; // Определение ресурса.

int cl::get_resource()
{
  if(resource) return 0; // Ресурс занят.
  else {
```

```
    resource = 1;
    return 1;  // Ресурс предоставлен данному объекту.
  }
}

int main()
{
  c1 ob1, ob2;

  /* Функция get_resource() является статической,
     поэтому ее можно вызывать независимо от объекта. */
  if(c1::get_resource()) cout << "Объект ob1 обладает ресурсом\n";

  if(!c1::get_resource()) cout << "Объекту ob2 доступ запрещен\n";

  ob1.free_resource();

  if(ob2.get_resource()) // can still call using object syntax
    cout << "Теперь объект ob2 может использовать ресурс\n";

  return 0;
}
```

Статические функции-члены имеют ограниченный круг приложений, однако они бывают полезны для предварительной инициализации закрытых статических переменных-членов до создания реальных объектов. Например, следующая программа абсолютно верна.

```
#include <iostream>
using namespace std;

class static_type {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_type::i; // определяем переменную i

int main()
{
  // Инициализируем статические данные перед созданием объекта
  static_type::init(100);

  static_type x;
  x.show(); // Отображаем число 100

  return 0;
}
```

Вызов конструкторов и деструкторов

Как правило, конструктор объекта вызывается при создании объекта, а деструктор — при его уничтожении. Точный смысл этих событий мы рассмотрим ниже.

Конструкторы локальных объектов последовательно вызываются при их объявлении. Деструкторы локальных объектов вызываются в обратном порядке.

Конструкторы глобальных объектов выполняются *до* функции **main()**. Глобальные конструкторы выполняются в порядке их объявления в файле. Порядок вызова глобальных конструкторов, распределенных в нескольких файлах, заранее не определен. Глобальные деструкторы вызываются в обратном порядке *после* завершения работы функции **main()**.

Вызовы конструкторов и деструкторов иллюстрируются следующей программой.

```
#include <iostream>
using namespace std;

class myclass {
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
  cout << "Инициализация " << id << "\n";
  who = id;
}

myclass::~myclass()
{
  cout << "Уничтожение " << who << "\n";
}

int main()
{
  myclass local_ob1(3);

  cout << "Эта строка уже не будет первой.\n";

  myclass local_ob2(4);

  return 0;
}
```

В результате на экране появятся следующие строки.

```
Инициализация 1
Инициализация 2
Инициализация 3
Эта строка уже не будет первой.
Инициализация 4
Уничтожение 4
Уничтожение 3
Уничтожение 2
Уничтожение 1
```

И еще: поскольку компиляторы и операционные системы отличаются друг от друга, последние две строки не всегда появляются на экране.

Оператор разрешения области видимости

Как известно, оператор "`::`" связывает между собой имена класса и его члена, сообщая компилятору, какому классу принадлежит данный член. Однако оператор разрешения области видимости имеет еще одно предназначение: он открывает доступ к переменной, имя которой "замаскировано" внутри вложенной области видимости. В качестве примера рассмотрим следующий фрагмент программы.

```
int i;  // Глобальная переменная i

void f()
{
  int i; // Локальная переменная i

  i = 10; // Используем локальную переменную i
  .
  .
  .
}
```

Как указано в комментарии, присваивание `i=10` относится к локальной переменной `i`. А что делать, если функции `f()` нужен доступ к глобальной переменной `i`? Тогда следует применить оператор разрешения области видимости.

```
int i;  // Глобальная переменная i.

void f()
{
  int i; // Локальная переменная i.

  ::i = 10; // Обращение к глобальной переменной i.
  .
  .
}
```

Вложенные классы

Один класс можно определить внутри другого. Таким образом создаются *вложенные классы*. Поскольку объявление класса фактически определяет его область видимости, вложенные классы могут существовать только внутри области видимости внешнего класса. Вообще-то, вложенные классы используются редко. Поскольку в языке C++ существует гибкий и мощный механизм наследования, необходимости создавать вложенные классы практически нет.

Локальные классы

Класс можно определить внутри функции. Рассмотрим следующий пример.

```
#include <iostream>
using namespace std;
```

```
void f();

int main()
{
  f();
  // Класс myclass отсюда не виден.
  return 0;
}

void f()
{
  class myclass {
    int i;
  public:
    void put_i(int n) { i=n; }
    int get_i() { return i; }
  } ob;

  ob.put_i(10);
  cout << ob.get_i();
}
```

На локальные классы налагается несколько ограничений. Во-первых, все функции-члены должны определяться внутри объявления локального класса. Во-вторых, локальные классы не могут обращаться к локальным переменным, объявленным внутри функции, за исключением статических локальных переменных (**static**) и внешних переменных (**extern**). Однако они имеют доступ к именам типов и перечислениям, определенным во внешней функции. В-третьих, внутри локальных классов нельзя объявлять статические переменные. Из-за этих ограничений локальные классы применяются довольно редко.

Передача объектов функциям

Объекты можно передавать функциям, как обычные переменные. Для этого применяется обычный механизм передачи параметров по значению. Несмотря на внешнюю простоту, этот процесс может привести к неожиданным последствиям, касающимся конструкторов и деструкторов. Попробуем разобраться в этом с помощью следующей программы.

```
// Передача объекта функции.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  myclass(int n);
  ~myclass();
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass::myclass(int n)
{
  i = n;
```

```
  cout << "Создание " << i << "\n";
}

myclass::~myclass()
{
  cout << "Уничтожение " << i << "\n";
}

void f(myclass ob);

int main()
{
  myclass o(1);

  f(o);
  cout << "Это переменная i в функции main: ";
  cout << o.get_i() << "\n";

  return 0;
}

void f(myclass ob)
{
  ob.set_i(2);

  cout << "Это локальная переменная i: " << ob.get_i();
  cout << "\n";
}
```

В результате работы этой программы на экране появятся следующие строки.

```
Создание 1
Это локальная переменная i: 2
Уничтожение 2
Это переменная i в функции main: 1
Уничтожение 1
```

Как показывают эти результаты, в ходе работы программы конструктор вызывался лишь однажды, при создании объекта **o** в функции **main()**, в то же время, деструктор был вызван *дважды*. Рассмотрим причины этой ситуации.

Когда объект передается в функцию, создается его копия. Именно эта копия становится параметром функции. Это означает, что в программе появился новый объект. По завершении работы функции копия аргумента (т.е. параметр) уничтожается. Это порождает два принципиально важных вопроса. Во-первых, вызывается ли конструктор при создании копии аргумента? Во-вторых, вызывается ли деструктор при уничтожении параметра? Ответы на эти вопросы вас удивят.

При создании копии аргумента обычный конструктор *не вызывается*. Вместо него вызывается *конструктор копирования*. Именно он и создает новую копию аргумента. Как указывается в главе 14, конструктор копирования можно определять явно. Однако, если объявление класса не содержит явного определения конструктора копирования, как в нашем примере, вызывается автоматический конструктор копирования, предусмотренный по умолчанию. Он создает побитовую (т.е. идентичную) копию объекта. Это легко понять. Ведь обычный конструктор нельзя вызывать для создания копии уже существующего объекта, поскольку такой вызов заменил бы текущее состояние объекта первоначальным. В то же время при

передаче объекта в функцию нам нужен полный дубликат объекта, отражающий его текущее, а не первоначальное состояние.

По завершении работы функции копия объекта выходит из области видимости, поэтому *вызывается* его деструктор. Вот почему в предыдущей программе деструктор вызывался дважды. Сначала он вызывается, когда параметр функции `f()` выходит из своей области видимости, а затем — когда при завершении работы программы уничтожается объект `o`.

Подведем итоги. Если возникает необходимость создать копию объекта и передать ее функции, обычный конструктор не вызывается. Вместо него вызывается автоматический конструктор копирования, создающий побитовый дубликат объекта. Однако при уничтожении копии объекта (как правило, в момент возвращения из функции) вызывается его деструктор.

Поскольку автоматический конструктор копирования создает точную копию оригинала, он может стать источником проблем. Даже если объект передается функции по значению, что теоретически позволяет защитить аргумент от изменения, возможны побочные эффекты. Например, если объект, передаваемый как параметр, выделяет динамическую память в момент своего создания и освобождает ее при своем уничтожении, его локальная копия внутри функции будет освобождать ту же самую область памяти при вызове деструктора. Это приведет к повреждению оригинала. Для того чтобы предотвратить появление таких проблем, в классе необходимо явно определить конструктор копирования (см. главу 14).

Возвращение объектов

Функция может возвращать объект вызывающему модулю. В качестве примера проанализируем следующую программу.

```
// Возвращение объектов из функций.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass f();  // Возвращение объекта класса myclass.

int main()
{
  myclass o;

  o = f();

  cout << o.get_i() << "\n";

  return 0;
}

myclass f()
{
  myclass x;
```

```
  x.set_i(1);
  return x;
}
```

При возвращении из функции автоматически создается временный объект, в котором хранится возвращаемое значение. Именно этот объект на самом деле возвращается в вызывающий модуль. После возвращения этот объект уничтожается. Это может привести к непредсказуемым последствиям. Например, если деструктор возвращаемого объекта должен освобождать динамическую память, она будет освобождаться, даже если объект, которому присваивается возвращаемое значение, по-прежнему ссылается на нее. Избежать этого можно с помощью перегрузки оператора присваивания (см. главу 15) и определения конструктора копирования (см. главу 14).

Присваивание объектов

Если объекты имеют одинаковый тип, их можно присваивать друг другу. При этом данные, содержащиеся в объекте, стоящем в правой части оператора присваивания, присваиваются переменным-членам объекта, стоящего в левой части. Проиллюстрируем этот механизм с помощью следующей программы, выводящей на экран число **99**.

```
// Присваивание объектов.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

int main()
{
  myclass ob1, ob2;

  ob1.set_i(99);
  ob2 = ob1; // Присваивание данных объекта ob1 объекту ob2.

  cout << "Это переменная i из объекта i: " << ob2.get_i();

  return 0;
}
```

По умолчанию все данные из одного объекта присваиваются соответствующим переменным другого объекта с помощью побитового копирования. Однако оператор присваивания можно перегрузить и определить другую процедуру (см. главу 15).

Полный справочник по

Глава 13

Массивы, указатели, ссылки и операторы динамического распределения памяти

В первой части книги мы уже изучили указатели и массивы, а также их применение ко встроенным типам данных. Теперь рассмотрим их взаимодействие с объектами. Кроме того, обсудим новое понятие — *ссылки*, тесно связанные с указателями. В завершение ознакомимся с операторами динамического распределения памяти.

Массивы объектов

В языке C++ массивы могут состоять из объектов. С синтаксической точки зрения объявление массива объектов ничем не отличается от объявления массива встроенного типа. Вот как выглядит программа, в которой используется массив, состоящий из трех элементов.

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  void set_i(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3];
  int i;

  for(i=0; i<3; i++) ob[i].set_i(i+1);

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

Эта программа выводит на экран числа **1**, **2** и **3**.

Если в классе определен конструктор с параметрами, каждый объект в массиве инициализируется с помощью списка инициализации, как это принято для массивов любого типа. Однако точный вид списка инициализации зависит от количества параметров конструктора. Если конструктор имеет лишь один параметр, можно просто задать список начальных значений, используя обычные синтаксические конструкции, предназначенные для инициализации массивов. По мере создания элементов массива эти значения поочередно присваиваются параметру конструктора. Вот как выглядит слегка измененная версия предыдущей программы.

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl(int j) { i=j; } // Конструктор
  int get_i() { return i; }
};
```

```
int main()
{
  cl ob[3] = {1, 2, 3}; // Список инициализации
  int i;

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

Как и прежде, эта программа выводит на экран числа **1**, **2** и **3**.

Фактически список инициализации, показанный в этой программе, является сокращенным вариантом более сложной формы:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Здесь конструктор **cl** вызывается явно. Разумеется, короткая форма инициализации используется чаще. В основе этого способа лежит возможность автоматического преобразования типа, если конструктор имеет только один аргумент (см. главу 12). Таким образом, сокращенную форму инициализации можно использовать, только если массив состоит из объектов, конструкторы которых имеют лишь один аргумент.

Если конструктор объекта имеет несколько аргументов, следует применять полную форму инициализации. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class cl {
  int h;
  int i;
public:
  cl(int j, int k) { h=j; i=k; } // Конструктор с двумя параметрами
  int get_i() {return i;}
  int get_h() {return h;}
};

int main()
{
  cl ob[3] =
  {
    cl(1, 2), // Инициализация
    cl(3, 4),
    cl(5, 6)
  };

  int i;

  for(i=0; i<3; i++)
  {
    cout << ob[i].get_h();
    cout << ", ";
    cout << ob[i].get_i() << "\n";
  }

  return 0;
}
```

Здесь конструктор массива **cl** имеет два аргумента. Следовательно, следует применять не сокращенную, а полную форму инициализации.

Инициализированные и неинициализированные массивы

Особая ситуация возникает, когда необходимо создать как инициализированные, так и неинициализированные массивы объектов. Рассмотрим следующий класс.

```
class cl {
  int i;
public:
  cl(int j) { i=j; } // Конструктор
  int get_i() { return i; }
};
```

Здесь в классе **cl** определен конструктор с одним параметром. Это значит, что любой массив такого типа должен быть инициализирован. Таким образом, массив нельзя объявить обычным образом.

```
cl a[9]; // Ошибка, конструктору необходим список инициализации
```

Этот оператор не работает, поскольку предполагается, что класс **cl** не имеет параметров, так как в объявлении массива не указан список инициализации. Однако класс **cl** не содержит конструкторов, не имеющих параметров. Итак, поскольку данному объявлению не соответствует ни один конструктор, компилятор выдаст сообщение об ошибке. Чтобы решить эту проблему, необходимо перегрузить конструктор, добавив вариант, не имеющий параметров, как показано ниже. Теперь в программе можно объявить оба вида массива.

```
class cl {
  int i;
public:
  cl() { i=0; } // Вызывается для неинициализированных массивов
  cl(int j) { i=j; } // Вызывается для инициализированных массивов
  int get_i() { return i; }
};
```

В таком варианте программы допускаются следующие операторы.

```
cl a1[3] = {3, 5, 6}; // Инициализированный массив

cl a2[34]; // Неинициализированный массив
```

Указатели на объекты

Указатели могут ссылаться не только на переменные встроенных типов, но и на объекты. Для доступа к членам класса через указатель на объект используется оператор "->", а не ".". Проиллюстрируем это с помощью следующей программы.

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob(88), *p;
```

```
  p = &ob; // Получаем адрес объекта ob.

  cout << p->get_i(); // для вызова функции get_i()
                      // применяется оператор ->

  return 0;
}
```

Как известно, при увеличении указателя на единицу он перемещается на следующий элемент того же типа. Например, целочисленный указатель будет ссылаться на следующее целое число. Как правило, адресная арифметика зависит от типа указателя. (Иначе говоря, она зависит от типа данных, на который ссылается указатель.) Это правило касается и указателей на объекты. Например, следующая программа использует указатель для доступа ко всем трем элементам массива **ob**.

```
#include <iostream>
using namespace std;

class cl {
  int i;
public:
  cl() { i=0; }
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3};
  cl *p;
  int i;

  p = ob; // Установить указатель на первый элемент массива.
  for(i=0; i<3; i++)
  {
    cout << p->get_i() << "\n";
    p++; // Указатель на следующий объект.
  }

  return 0;
}
```

Указателю можно присвоить адрес открытого члена объекта, а затем ссылаться на этот член с его помощью. Рассмотрим следующую программу, которая выводит на экран число **1**.

```
#include <iostream>
using namespace std;

class cl {
public:
  int i;
  cl(int j) { i=j; }
};

int main()
{
  cl ob(1);
  int *p;
```

```
  p = &ob.i;  // Получить адрес члена ob.i.

  cout << *p; // Обращение к члену ob.i через указатель p

  return 0;
}
```

Поскольку указатель **p** ссылается на целое число, он имеет тип **int**. В данном случае не имеет значения, что переменная **i** является членом объекта **ob**.

Проверка типа указателей

Работая с указателями, следует иметь в виду: присваивать можно лишь указатели совместимых типов.

```
int *pi;
float *pf;
```

Следующий оператор неверен.

```
pi = pf; // Ошибка — несовместимость типов.
```

Разумеется, любую несовместимость можно преодолеть, используя приведение типов, однако это нарушает принципы строгой проверки типов.

Указатель this

При вызове функции-члена ей неявно передается указатель на вызывающий объект. Этот указатель называется **this**. Рассмотрим программу, в которой описан класс **pwr**, предназначенный для вычисления степени некоторого числа.

```
#include <iostream>
using namespace std;

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr() { return val; }
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);

  cout << x.get_pwr() << " ";
```

```
  cout << y.get_pwr() << " ";
  cout << z.get_pwr() << "\n";

  return 0;
}
```

Внутри класса к функции-члену можно обращаться напрямую, не используя объекты и название класса. Таким образом, внутри конструктора **pwr()** оператор

```
b = base;
```

означает, что переменной **b**, принадлежащей вызывающему объекту, присваивается значение переменной **base**. Однако тот же самый оператор можно переписать иначе.

```
this->b = base;
```

Указатель **this** ссылается на объект, вызывающий функцию **pwr()**. Таким образом, выражение **this->b** ссылается на переменную **b**, принадлежащую текущему объекту. Например, если функция **pwr()** вызвана объектом **x** (в объявлении **x(4.0, 2)**), то указатель **this** в предыдущем операторе будет ссылаться на объект **x**. Впрочем, этот оператор можно записать в сокращенном виде, не используя указатель **this**.

Рассмотрим полное определение конструктора **pwr()**, написанное с помощью указателя **this**.

```
pwr::pwr(double base, int exp)
{
  this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--)
    this->val = this->val * this->b;
}
```

На самом деле ни один программист на языке C++ не станет писать конструктор таким образом, поскольку сокращенная форма намного проще. Однако указатель **this** очень важен при перегрузке операторов, а также в ситуациях, когда функция-член должна использовать указатель на вызывающий объект.

Помните, что указатель **this** автоматически передается всем функциям-членам. Следовательно, функцию **get_pwr()** можно переписать иначе.

```
double get_pwr() { return this->val; }
```

В этом случае, если функция **get_pwr()** вызывается с помощью оператора

```
y.get_pwr();
```

указатель **this** будет ссылаться на объект **y**.

В заключение сделаем два важных замечания. Во-первых, дружественные функции не являются функциями-членами, и, следовательно, им не передается указатель **this**. Во-вторых, статические функции-члены также не получают указатель **this**.

Указатели на производные типы

Как правило, указатель одного типа не может ссылаться на объект другого типа. Однако у этого правила есть важное исключение, касающееся производных классов. Предположим для начала, что в программе объявлены два класса: **B** и **D**. Кроме того,

допустим, что класс `D` является производным от базового класса `B`. В этом случае указатель типа `B*` может ссылаться и на объекты типа `D`. В принципе, указатель на базовый класс можно использовать как указатель на объект любого производного класса.

Однако обратное утверждение неверно. Указатель типа `D*` не может ссылаться на объекты класса `B`. Кроме того, с помощью указателя на базовый класс можно ссылаться только на наследуемые члены, но не на новые члены производного класса. (Однако указатель на базовый класс можно привести к типу указателя на производный класс и получить доступ ко всем членам производного класса.)

Рассмотрим короткую программу, иллюстрирующую применение указателя на базовый класс для доступа к объектам производного класса.

```
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) { j=num; }
  int get_j() { return j; }
};

int main()
{
  base *bp;
  derived d;

  bp = &d; // Базовый указатель ссылается на объект
           // производного класса.

  // Доступ к объекту производного класса с помощью
  // указателя на производный класс.
  bp->set_i(10);
  cout << bp->get_i() << " ";

/* Следующий оператор не работает. На элементы производного
   класса нельзя ссылаться с помощью указателя на базовый
   класс.

   bp->set_j(88); // Ошибка
   cout << bp->get_j(); // Ошибка

*/
  return 0;
}
```

Как видим, указатель на базовый класс позволяет обращаться к объекту производного класса.

Выше мы уже отмечали, что указатель на базовый класс можно привести к типу указателя на производный класс. Проиллюстрируем эту возможность следующим примером.

```
// Приведение типа открывает доступ
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

Следует помнить, что адресная арифметика зависит от типа базового указателя. По этой причине, если указатель на объекты базового класса используется для доступа к производным объектам, увеличение указателей не откроет доступ к следующему объекту производного типа. Вместо этого он будет ссылаться на объект базового типа. Разумеется, это может вызвать недоразумения. Рассмотрим программу, которая, будучи совершенно правильной с синтаксической точки зрения, является ошибочной.

```
// Эта программа содержит ошибку.
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};

int main()
{
  base *bp;
  derived d[2];

  bp = d;

  d[0].set_i(1);
  d[1].set_i(2);

  cout << bp->get_i() << " ";
  bp++; // Этот оператор относится к базовому,
        // а не производному типу
  cout << bp->get_i(); // На экран выводится мусор.

  return 0;
}
```

Использование указателей на производные типы оказывается полезным для поддержки динамического полиморфизма с помощью механизма виртуальных функций (см. главу 17).

Указатели на члены класса

В языке C++ существует особый тип указателя, который ссылается на член класса вообще, а не на конкретный экземпляр этого члена в объекте. Указатель такого вида называется *указателем на член класса*. Этот необычный указатель задает смещение

внутри объекта соответствующего класса. Поскольку указатели на члены класса не являются указателями в обычном смысле слова, к ним нельзя применять операторы "." и "->". Чтобы обратиться к члену класса с помощью указателя на него, следует применять особые операторы: ".*" и "->*".

Рассмотрим пример.

```
#include <iostream>
using namespace std;

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // Указатель на член класса
  int (cl::*func)(); // Указатель на функцию-член
  cl ob1(1), ob2(2); // Создаем объекты

  data = &cl::val; // Определяем смещение члена val
  func = &cl::double_val; // Определяем смещение функции double_val()

  cout << "Значения: ";
  cout << ob1.*data << " " << ob2.*data << "\n";

  cout << "Удвоенные значения: ";
  cout << (ob1.*func)() << " ";
  cout << (ob2.*func)() << "\n";

  return 0;
}
```

Эта программа создает два указателя на члены класса: **data** и **func**. Обратите особое внимание на синтаксические особенности их объявлений. Объявляя указатели на члены, следует задавать имя класса и применять оператор разрешения области видимости. Кроме того, программа создает два объекта класса **cl**: **ob1** и **ob2**. Указатели на члены класса могут ссылаться как на переменные, так и на члены. Затем вычисляются адреса членов **val** и **double_val()**. Эти "адреса" представляют собой смещения соответствующих членов в объекте класса **cl**. Значения, хранящиеся в переменной **val** в каждом из объектов, выводятся на экран с помощью указателя **data**. В заключение программа вызывает функцию **double_func()**, используя переменную **func**, являющуюся указателем на член класса. Обратите внимание на то, что для правильного выполнения оператора ".*" необходимы дополнительные скобки.

Для доступа к члену класса через объект или ссылку на него используется оператор ".*". Если задан указатель на объект, для доступа к его членам необходимо применять оператор "->*". Проиллюстрируем сказанное следующим примером.

```
#include <iostream>
using namespace std;

class cl {
public:
  cl(int i) { val=i; }
```

```
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // Указатель на переменную-член
  int (cl::*func)(); // Указатель на функцию-член
  cl ob1(1), ob2(2); // Создаем объекты
  cl *p1, *p2;

  p1 = &ob1; // Доступ к объекту через указатель
  p2 = &ob2;

  data = &cl::val; // Определяем смещение переменной val
  func = &cl::double_val; // Определяем смещение функции double_val()

  cout << "Значения: ";
  cout << p1->*data << " " << p2->*data << "\n";

  cout << "Удвоенные значения: ";
  cout << (p1->*func)() << " ";
  cout << (p2->*func)() << "\n";

  return 0;
}
```

В этом варианте программы переменные **p1** и **p2** являются указателями на объекты класса **cl**, поэтому для доступа к членам **val** и **double_func()** применяется оператор "**->***".

Запомните, что указатели на члены отличаются от указателей на конкретные элементы объекта. Рассмотрим фрагмент программы, полагая, что класс **cl** объявлен, как показано выше.

```
int cl::*d;
int *p;
cl o;

p = &o.val // Адрес конкретной переменной val

d = &cl::val // Смещение обобщенной переменной val
```

Здесь указатель **p** ссылается на целочисленную переменную, принадлежащую конкретному объекту. В то же время переменная **d** хранит смещение члена **val** внутри любого объекта класса **cl**.

Как правило, операторы, связанные с указателями на члены класса, применяются в исключительных ситуациях. В повседневном программировании они обычно не используются.

Ссылки

В языке C++ есть переменные, очень напоминающие указатели. Они называются *ссылками* (reference). По существу, ссылка — это неявный указатель. Они используются: для передачи параметров функции, для возврата значения функции и в качестве самостоятельных переменных. Рассмотрим каждый из этих вариантов.

Передача параметров с помощью ссылок

Вероятно, самое важное применение ссылок заключается в создании функций, которые автоматически предусматривают передачу параметров по ссылке. Как указывалось в главе 3, аргументы передаются функции двумя способами: по значению и по ссылке. Если аргумент передается по значению, функция получает его копию. По умолчанию в языке C++ применяется передача аргументов по значению, однако есть две возможности передать их по ссылке. Во-первых, можно явно передать указатель на аргумент. Во-вторых, можно передать ссылку на аргумент. В большинстве ситуаций второй способ более предпочтителен.

Чтобы осознать, насколько ценной является эта возможность, рассмотрим механизм передачи параметров по ссылке с помощью указателей. Следующая программа явно создает указатель на аргумент функции **neg()**, предназначенной для изменения знака целочисленной переменной.

```
// Аргумент передается по ссылке с помощью явного указателя.
#include <iostream>
using namespace std;

void neg(int *i);

int main()
{
  int x;

  x = 10;
  cout << x << " является отрицанием числа ";

  neg(&x);
  cout << x << "\n";

  return 0;
}

void neg(int *i)
{
  *i = -*i;
}
```

В данной программе функция **neg()** получает в качестве параметра указатель на целочисленную переменную, знак которой следует поменять на противоположный. Следовательно, функция **neg()** должна явно получать адрес переменной **x**. Кроме того, чтобы получить доступ к переменной, на которую ссылается указатель **i** внутри функции **neg()**, необходимо применять оператор "*". Вот как устроен механизм передачи аргументов по ссылке в языке C++, причем в языке C этот способ является единственным. К счастью, в языке C++ существует возможность автоматизировать этот механизм, используя параметры, являющиеся ссылками.

Чтобы создать ссылку на параметр, перед его именем следует ставить символ **&**. Вот как, например, объявляется ссылка на аргумент **i** функции **neg()**.

```
void neg(int &i);
```

В этом случае имя **i** становится вторым именем аргумента функции **neg()**, и любые операции над ней автоматически распространяются на фактический аргумент. С технической точки зрения переменная **i** является неявным указателем на аргумент функции. Теперь оператор "*" становится лишним. Кроме того, теперь необязательно ставить перед именем аргумента символ **&**. Рассмотрим новую версию программы.

```
// Применение ссылки на аргумент.
#include <iostream>
using namespace std;

void neg(int &i); // Переменная i стала ссылкой

int main()
{
  int x;

  x = 10;
  cout << x << " является отрицанием числа ";

  neg(x); // Оператор & больше не нужен
  cout << x << "\n";

  return 0;
}

void neg(int &i)
{
  i = -i; // Переменная i является ссылкой, оператор * больше не
нужен
}
```

Ссылка на параметр автоматически (неявно) устанавливается на аргумент функции. Следовательно, в предыдущей программе оператор

```
i = -i ;
```

на самом деле работает с переменной **x**, а не с ее копией. Оператор **&** становится не нужен. Кроме того, внутри функции параметр, переданный с помощью ссылки, используется непосредственно, без применения оператора "*". Как правило, значение, присвоенное ссылке, на самом деле присваивается переменной, к которой эта ссылка относится.

Внутри функции ссылку на параметр невозможно связать с другой переменной. Иначе говоря, оператор

```
i++;
```

внутри функции **neg()** увеличивает значение параметра, а не перемещает указатель **i** на новую ячейку.

Рассмотрим еще один классический пример. Эта программа использует ссылки на параметры для их перестановки внутри функции **swap()**.

```
#include <iostream>
using namespace std;

void swap(int &i, int &j);

int main()
{
  int a, b, c, d;

  a = 1;
  b = 2;
  c = 3;
  d = 4;

  cout << "a и b: " << a << " " << b << "\n";
  swap(a, b); // Оператор & не нужен
```

```
  cout << "a и b: " << a << " " << b << "\n";

  cout << "c и d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c и d: " << c << " " << d << "\n";

  return 0;
}

void swap(int &i, int &j)
{
  int t;

  t = i; // Оператор * не нужен
  i = j;
  j = t;
}
```

В результате работы этой программы на экран будут выданы следующие строки.

```
a и b: 1 2
a и b: 2 1
c и d: 3 4
c и d: 4 3
```

Передача ссылок на объекты

В главе 12 указывалось, что при передаче объекта в качестве параметра функции создается его копия. По завершении работы функции вызывается деструктор копии. Однако, если параметр вызывается по ссылке, копия объекта не создается. Это значит, что после возврата управления параметр функции не уничтожается и деструктор не вызывается. Рассмотрим следующий пример.

```
#include <iostream>
using namespace std;

class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl();
  void neg(cl &o) { o.i = -o.i; } // Временный объект не создается
};

cl::cl(int num)
{
  cout << "Создание объекта " << num << "\n";
  id = num;
}

cl::~cl()
{
  cout << "Уничтожение объекта " << id << "\n";
}

int main()
{
  cl o(1);
```

```
  o.i = 10;
  o.neg(o);

  cout << o.i << "\n";

  return 0;
}
```

Программа выводит на экран следующие результаты.

```
Создание объекта 1
-10
Уничтожение объекта 1
```

Как видим, деструктор класса **cl** вызывается лишь один раз. Если бы объект **o** передавался по значению, внутри функции **neg()** был бы создан второй объект, для уничтожения которого пришлось бы вызывать деструктор.

В функции **neg()** для доступа к члену объекта используется оператор "**.**", поскольку оператор "**->**" применяется лишь к указателям.

При передаче параметров по ссылке следует помнить, что все изменения, происходящие с параметром внутри функции, отражаются в вызывающем модуле.

И, наконец, передача по ссылке происходит быстрее, чем передача по значению. Исключением из этого правила могут быть лишь микроскопические объекты. Аргументы, как правило, помещаются в стек. Вследствие этого процесс заталкивания больших объектов в стек и выталкивания их оттуда занимает много времени.

Возврат ссылок

Функция может возвращать ссылку на объект. Таким образом, вызов функции может стоять в левой части оператора присваивания! Рассмотрим следующий пример.

```
#include <iostream>
using namespace std;

char &replace(int i); // Возврат ссылки

char s[80] = "Общий привет";

int main()
(

  replace(5) = 'X'; // Вставляем символ X между словами
                    // "Общий" и "привет"
  cout << s;

  return 0;
}

char &replace(int i)
{
  return s[i];
}
```

Эта программа вставляет символ X вместо пробела между словами "Общий" и "привет", т.е. выводит на экран строку "ОбщийXпривет". Рассмотрим, как это происходит. Во-первых, функция **replace()** возвращает ссылку на символ, входящий в строку **s**, индекс которого задан переменной **i**. Затем

в функции `main()` с помощью этой ссылки соответствующему элементу строки присваивается символ `X`.

При возврате ссылки следует иметь в виду, что по завершении работы функции возвращаемый объект не выходит из области видимости.

Независимые ссылки

Мы рассмотрели две ситуации, в которых применяются ссылки: передача параметров и возврат значения функции. Однако ссылку можно объявлять просто как переменную. Такие ссылки называются *независимыми*.

Независимые ссылки являются псевдонимами объектов. При объявлении независимых ссылок они обязательно инициализируются, поскольку после инициализации их нельзя связывать с другими объектами. Следовательно, единственный способ задать значение ссылки — инициализировать ее. (В языке C++ инициализация значительно отличается от оператора присваивания.)

Рассмотрим программу, в которой применяются независимые ссылки.

```
#include <iostream>
using namespace std;

int main()
{
  int a;
  int &ref = a; // Независимая ссылка

  a = 10;
  cout << a << " " << ref << "\n";

  ref = 100;
  cout << a << " " << ref << "\n";

  int b = 19;
  ref = b; // Присваиваем переменной a значение переменной b
  cout << a << " " << ref << "\n";

  ref--; // Уменьшение на единицу переменной a.

  cout << a << " " << ref << "\n";

  return 0;
}
```

Результат работы программы имеет следующий вид.

```
10 10
100 100
19 19
18 18
```

Фактически независимые ссылки не играют значительной роли в программировании, поскольку они представляют собой лишь второе имя другой переменной, а объекты, имеющие несколько имен, свидетельствуют о плохой организации программы.

Ссылки на производные типы

Ссылка на базовый класс, как и указатель, может использоваться для доступа к объектам производного класса. Чаще всего эта возможность используется при передаче параметров функций. Параметру, передаваемому с помощью ссылки на базовый класс, можно присваивать объекты как базового, так и производных классов.

Ограничения на ссылки

В языке C++ существуют ограничения на работу со ссылками. Например, ссылки не могут связываться с другими ссылками. Иначе говоря, ссылка не имеет адреса. Кроме того, невозможно создать указатель на ссылку. К битовому полю также нельзя обратиться с помощью ссылки.

Независимые ссылки должны быть инициализированы в момент своего объявления. Нулевой ссылки не существует.

Стиль

Объявляя указатели и ссылки, некоторые программисты используют единообразный стиль, связывая символы * и & с именем типа, а не с переменной. Рассмотрим два эквивалентных объявления.

```
int& p; // Символ & связан с типом
int &p; // Символ & связан с переменной
```

Связывание символов * или & с типом отражает желание некоторых программистов создать отдельный тип указателей. Однако это может вызвать недоразумения, поскольку символы * и & не распространяются на список переменных. Например, в следующем объявлении создается *один* целочисленный указатель, а не *два*.

```
int* a, b;
```

Здесь **b** — целочисленная переменная, а не указатель, поскольку в соответствии с правилами языка C++ символ * (как и символ &) связывается с отдельной переменной, а не с ее типом. Причина недоразумения заключается в том, что визуально это объявление выглядит так, будто в нем объявляются два целочисленных указателя, в то время как указателем является лишь переменная **a**. Это объявление может ввести в заблуждение не только новичка, но и опытного программиста.

Следует четко понимать, что компилятору абсолютно безразлично, как написано объявление: **int *p** или **int* p**. Программист может выбирать свой стиль. Однако мы будем связывать символы & и * именно с переменными, чтобы избежать недоразумений.

Операторы динамического распределения памяти

В языке C++ предусмотрены два оператора динамического распределения памяти: **new** и **delete**. Эти операторы выделяют и освобождают память в ходе выполнения программы. Динамическое распределение памяти представляет собой важную часть практически всех реальных программ. Как указывалось в первой части книги, в языке C++ есть альтернативный способ динамического распределения памяти, основанный на функциях **malloc()** и **free()**. Они сохранены для того, чтобы достичь совместимости с языком C. Однако в программах на языке C++ следует применять операторы **new** и **delete**, поскольку они имеют ряд важных преимуществ.

Оператор **new** выделяет область памяти и возвращает указатель на ее первую ячейку. Оператор **delete** освобождает память, выделенную ранее с помощью оператора **new**. Общий вид этих операторов таков.

```
указатель = new тип;
delete указатель;
```

Здесь *указателю* присваивается адрес первой ячейки выделенной области памяти, размер которой определяется *типом*.

Поскольку объем кучи ограничен, память может оказаться исчерпанной. В этом случае оператор **new** сгенерирует исключительную ситуацию **bad_alloc**, определенную в заголовке **<new>**. Программа должна перехватить эту ситуацию и обработать ее. (Обработка исключительных ситуаций рассматривается в главе 19.) Если в программе не предусмотрена обработка исключительной ситуации **bad_alloc**, ее выполнение будет прервано.

Действие оператора **new** в исключительных ситуациях определено стандартом языка C++. Проблема заключается в том, что не все компиляторы полностью соответствуют стандарту. В ранних версиях языка C++ оператор **new** в случае отказа возвращал нулевой указатель. Позднее он стал генерировать исключительную ситуацию. В итоге было решено, что по умолчанию оператор **new** должен генерировать исключительную ситуацию, однако возвращать нулевой указатель также не запрещается. Следовательно, в каждом конкретном случае поведение оператора **new** регламентируется разработчиками компилятора. Разумеется, в конце концов все компиляторы будут вынуждены придерживаться стандарта, но пока следует ориентироваться на их документацию.

Все примеры, описанные в нашей книге, написаны в соответствии со стандартом. Если ваш компилятор не полностью соответствует стандарту, программы следует немного изменить.

Рассмотрим пример, в котором выделяется динамическая память для целочисленной переменной.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p;

  try {
    p = new int; // Выделить память для целочисленной переменной
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    return 1;
  }

  *p = 100;

  cout << "По адресу " << p << " ";
  cout << "записано значение " << *p << "\n";

  delete p;

  return 0;
}
```

Эта программа присваивает переменной **p** адрес динамической памяти, в которой может храниться целочисленная переменная. Затем в эту область памяти записывается число 100, а ее содержимое выводится на экран. В заключение программа освобождает выделенную память. Учтите, если ваш компилятор реализует оператор **new** так, что в случае отказа он возвращает нулевой указатель, эту программу придется модифицировать.

Оператор **delete** следует применять только к результату оператора **new**. В противном случае могут возникнуть проблемы, например, крах операционной системы.

Операторы **new** и **delete** аналогичны функциям **malloc()** и **free()**, но по сравнению с ними обладают несколькими преимуществами. Во-первых, оператор **new** автоматически выделяет количество памяти, необходимое объекту указанного типа. Оператор **sizeof** теперь применять не следует. Поскольку размер выделяемой памяти вычисляется автоматически, ошибки исключены. Во-вторых, оператор **new** автоматически возвращает указатель заданного типа. Теперь не нужно выполнять приведение типа, как это было при использовании функции **malloc()**. Операторы **new** и **delete** можно перегружать, что позволяет создавать настраиваемые системы для выделения динамической памяти.

Хотя формальных правил на этот счет не существует, операторы **new** и **delete** не следует смешивать с функциями **malloc()** и **free()** в рамках одной и той же программы, поскольку они могут оказаться несовместимыми.

Инициализация выделяемой памяти

Выделяемую память можно проинициализировать заданным значением. Для этого следует указать начальное значение после имени типа в операторе **new**. Общий вид оператора **new** в этом случае выглядит следующим образом.

```
указатель = new тип (начальное_значение)
```

Разумеется, тип начального значения должен быть совместимым с типом данных, для которых выделяется память.

Рассмотрим программу, которая записывает в выделенную память число 87.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p;

  try {
    p = new int (87); // Инициализируем числом 87
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    return 1;
  }

  cout << "По адресу " << p << " ";
  cout << "записано число " << *p << "\n";

  delete p;

  return 0;
}
```

Выделение памяти для массивов

С помощью оператора **new** можно выделять память для массивов. В этом случае оператор **new** имеет следующий вид.

```
указатель = new тип_массива[размер];
```

Здесь размер задает количество элементов размещаемого массива.

Для освобождения памяти, занятой массивом, применяется следующая форма оператора **delete**:

```
delete [] указатель;
```

Например, следующая программа выделяет память для целочисленного массива, состоящего из 10 элементов.

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  try {
    p = new int [10]; // Выделяем память для целочисленного
                      // массива, состоящего из 10 элементов
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    return 1;
  }

  for(i=0; i<10; i++ )
    p[i] = i;

  for(i=0; i<10; i++)
    cout << p[i] << " ";

  delete [] p; // Освобождаем память, занятую массивом

  return 0;
}
```

Обратите внимание на оператор **delete**. Как уже указывалось, при освобождении памяти, выделенной для массива с помощью оператора **new**, размер массива не указывается. (Как будет показано в следующем разделе, это свойство особенно важно при работе с массивами объектов.)

Выделение памяти для объектов

Используя оператор **new**, можно динамически выделять память для объектов. В этом случае оператор вернет указатель на созданный объект. Динамически созданный объект ничем не отличается от других. При его создании также вызывается конструктор (если он предусмотрен), а при освобождении памяти вызывается соответствующий деструктор.

Рассмотрим небольшую программу, в которой определен класс **balance**, предназначенный для хранения имени человека и суммы, лежащей на его банковском счету. Внутри функции **main()** объект класса **balance** создается динамически.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
  char name[80];
```

```
public:
  void set(double n, char *s)
  {
    cur_bal = n;
    strcpy(name, s);
  }

  void get_bal(double &n, char *s)
  {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;

  try {
    p = new balance;
  } catch (bad_alloc xa){
    cout << "Исключительная ситуация\n";
    return 1;
  }

  p->set(12387.87, "Ральф Уилсон");

  p->get_bal(n, s);

  cout << s << ": сумма = " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

Поскольку переменная **p** является указателем на объект, для доступа к его членам используется оператор "->".

Как известно, динамические объекты могут содержать конструкторы и деструктор. Кроме того, конструкторы могут иметь параметры. Рассмотрим модификацию предыдущей программы.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
```

```
  ~balance() {
    cout << "Уничтожение объекта ";
    cout << name << "\n";
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;

  // В этой версии используется инициализация
  try {
    p = new balance (12387.87, "Ральф Уилсон");
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << ": сумма = " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

Обратите внимание на то, что параметры конструктора объекта указаны после имени типа, как при обычной инициализации.

В динамической памяти можно разместить массивы объектов, однако следует иметь в виду одну ловушку. Массивы, размещенные в динамической памяти с помощью оператора **new**, нельзя инициализировать, поэтому следует убедиться, что класс содержит конструктор, не имеющий параметров. В противном случае компилятор языка C++ не найдет подходящего конструктора и при попытке разместить массив в динамической памяти выдаст сообщение об ошибке.

В следующей версии нашей программы в динамической памяти размещается массив **balance**, и вызывается конструктор, не имеющий параметров.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
```

```
  balance() {} // Конструктор без параметров
  ~balance() {
    cout << "Уничтожение объекта ";
    cout << name << "\n";
  }
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
  int i;

  try {
    p = new balance [3]; // Размещение массива
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    return 1;
  }

  // Используется оператор ., а не ->
  p[0].set(12387.87, "Ральф Уилсон");
  p[1].set(144.00, "А. С. Коннерс");
  p[2].set(-11.23, "И. М. Овердроун");

  for(i=0; i<3; i++) {
    p[i].get_bal(n, s);

    cout << s << ": сумма = " << n;
    cout << "\n";
  }

  delete [] p;
  return 0;
}
```

В результате работы этой программы на экран будут выведены следующие строки.

```
Ральф Уилсон: сумма = 12387.9
А. С. Коннерс: сумма = 144
И. М. Овердроун: сумма = -11.23
Уничтожение объекта И. М. Овердроун
Уничтожение объекта А. С. Коннерс
Уничтожение объекта Ральф Уилсон
```

Для уничтожения массива динамических объектов следует применять оператор **delete []**, чтобы можно было вызывать деструктор для каждого объекта отдельно.

Альтернатива nothrow

В случае нехватки памяти оператор **new** может не генерировать исключительную ситуацию, а возвращать нулевой указатель. Этот вид оператора **new** оказывается весьма полезным при работе со старыми компиляторами. Особенно ценно, что вызовы функции **malloc()** можно заменить оператором **new**. (Это часто приходится делать при переводе программ с языка С на язык С++.) Для этого применяется следующая форма оператора **new**.

```
указатель = new(nothrow) тип;
```

Эта форма оператора **new** напоминает его ранние версии. Поскольку в случае нехватки памяти он не генерирует исключительную ситуацию, а возвращает нулевой указатель, его можно внедрять в старые программы. Однако в новых программах лучше использовать новый вариант оператора **new**, генерирующий исключительную ситуацию **bad_alloc**. Чтобы иметь возможность использовать опцию **nothrow**, следует включить в программу заголовок **<new>**.

Рассмотрим программу, демонстрирующую альтернативу **new(nothrow)**.

```
// Демонстрация альтернативы new(nothrow).
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  p = new(nothrow) int[32]; // Применение опции nothrow
  if(!p) {
    cout << "Исключительная ситуация.\n";
    return 1;
  }

  for(i=0; i<32; i++) p[i] = i;

  for(i=0; i<32; i++) cout << p[i] << " ";

  delete [] p; // Освобождаем память

  return 0;
}
```

Как показывает эта программа, при использовании альтернативы **nothrow** следует проверять указатель, возвращаемый оператором **new**.

Буферизованный оператор new

В языке С++ существует особая форма оператора **new**, которую можно применять для указания альтернативного способа распределения динамической памяти. Эта форма называется *буферизованным оператором* **new** (replacement form of new). Наиболее полезным это средство оказывается при перегрузке оператора **new** в особых ситуациях. Его общий вид таков.

```
указатель = new (список_аргументов) тип;
```

Здесь *список аргументов* представляет собой перечисление значений, разделенных запятыми, которые передаются перегруженному оператору **new**.

Полный справочник по

Глава 14

Перегрузка функций, конструкторы копирования и аргументы по умолчанию

В этой главе рассматриваются перегрузка функций, конструкторы копирования и аргументы, задаваемые по умолчанию. Перегрузка функций — один из основных аспектов языка C++. Она обеспечивает статический полиморфизм, а также гибкость и комфорт. Чаще всего перегрузке подвергаются конструкторы. Одной из наиболее важных форм перегрузки конструкторов является конструктор копирования. Возможность задавать аргументы по умолчанию используют в качестве альтернативы перегрузке функций.

Перегрузка функций

Перегрузка функций — это использование одного имени для нескольких функций. Секрет перегрузки заключается в том, что каждое переопределение функции должно использовать либо другие типы параметров, либо другое их количество. Только эти различия позволяют компилятору определять, какую функцию следует вызвать в том или ином случае. Например, в следующей программе функция **myfunc()** перегружена для разных типов параметров.

```
#include <iostream>
using namespace std;

int myfunc(int i); // Эти варианты различаются типами параметров
double myfunc(double i);

int main()
{

  cout << myfunc(10) << " "; // Вызов функции myfunc(int i)
  cout << myfunc(5.4); // Вызов функции myfunc(double i)

  return 0;
}

double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}
```

В следующей программе перегруженные варианты функции **myfunc()** используют разное количество параметров.

```
#include <iostream>
using namespace std;

int myfunc(int i); // Эти варианты различаются количеством параметров
int myfunc(int i, int j);

int main()
{
  cout << myfunc(10) << " "; // Вызов функции myfunc(int i)
  cout << myfunc(4, 5); // Вызов функции myfunc(int i, int j)
```

```
  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

Следует помнить, что перегруженные функции должны отличаться типами или количеством параметров. Тип возвращаемого значения не позволяет перегружать функции. Например, следующий вариант перегрузки функции **myfunc()** неверен.

```
int myfunc(int i);   // Ошибка: разных типов возвращаемого float my-
func(int i); // значения недостачно для перегрузки
```

Иногда объявления двух функций внешне отличаются, но фактически совпадают. Рассмотрим в качестве примера следующие объявления.

```
void f(int *p);
void f(int p[]); // Ошибка, выражения *p и p[] эквивалентны
```

Следует помнить, что компилятор не различает выражения ***p** и **p[]**. Следовательно, хотя внешне два прототипа функции **f** различаются, на самом деле они полностью совпадают.

Перегрузка конструкторов

Конструкторы можно перегружать. Фактически их-то и перегружают чаще всего. Для перегрузки конструктора существуют три причины: гибкость, возможность создания инициализированных (неинициализированных) объектов и конструкторов копирования. В этом разделе мы рассмотрим только первые две причины. Конструктор копирования описывается в следующем разделе.

Перегрузка конструктора для достижения гибкости

Довольно часто объекты класса можно создать несколькими способами. Для каждого из этих способов можно определить отдельный вариант перегруженного конструктора. Эти варианты исчерпывают все возможности создать объект — при попытке сделать это непредусмотренным способом компилятор не найдет подходящего конструктора и выдаст сообщение об ошибке.

Перегруженные конструкторы намного повышают гибкость класса. Они позволяют пользователю выбирать оптимальный способ создания объекта. Рассмотрим программу, создающую класс **date** для хранения календарной даты. Обратите внимание на то, что конструктор перегружен дважды.

```
#include <iostream>
#include <cstdio>
using namespace std;
```

```
class date {
  int day, month, year;
public:
  date(char *d);
  date(int m, int d, int y);
  void show_date();
};

// Инициализация строкой.
date::date(char *d)
{
  sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
}

// Инициализация целыми числами.
date::date(int m, int d, int y)
{
  day = d;
  month = m;
  year = y;
}

void date::show_date()
{
  cout << month << "/" << day;
  cout << "/" << year << "\n";
}

int main()
{
  date ob1(12, 4, 2001), ob2("10/22/2001");

  ob1.show_date();
  ob2.show_date();

  return 0;
}
```

В этой программе объект класса **date** можно инициализировать двумя способами: задав месяц, день и год в виде трех целых чисел либо в виде строки *mm/dd/yyyy*. Оба способа применяются довольно часто, поэтому имеет смысл предусмотреть два разных конструктора для создания объектов класса **date**.

Этот пример иллюстрирует основную идею, лежащую в основе перегруженных конструкторов: они позволяют выбирать способ создания объектов, лучше всех соответствующий конкретной ситуации. Например, пользователь может ввести дату в виде массива **s**. Эту строку можно сразу использовать для создания объекта класса **date**. Для этого совершенно не требуется преобразовывать ее в другой вид. Однако, если бы конструктор **date()** не был бы перегружен, строку **s** пришлось бы разбить на три целых числа.

```
int main()
{
  char s[80];

  cout << "Введите новую дату: ";
  cin >> s;

  date d(s);
  d.show_date();
```

```
  return 0;
}
```

В другой ситуации пользователю удобнее инициализировать объект класса **date** тремя целыми числами. Например, если дата является результатом неких вычислений, более естественно создавать объект класса **date** с помощью конструктора **date(int, int, int)**. Итак, в каждом конкретном случае существует оптимальный вариант. Перегруженный конструктор обеспечивает необходимую гибкость класса, особенно необходимую при создании библиотек.

Создание инициализированных и неинициализированных объектов

Перегруженные конструкторы часто используются для создания инициализированных и неинициализированных объектов (точнее, инициализированных по умолчанию). Это особенно важно при создании динамических массивов объектов, которые, как известно, невозможно проинициализировать. Чтобы создать неинициализированный массив объектов и оставить возможность создания объектов, которым присвоено некое начальное значение, следует предусмотреть два варианта конструктора: с инициализацией и без нее.

Рассмотрим в качестве примера программу, в которой объявляются два массива типа **powers**. Один из них инициализируется, а другой — нет. Кроме того, эти объекты хранятся в динамическом массиве.

```
#include <iostream>
#include <new>
using namespace std;

class powers {
  int x;
public:
  // Перегрузка конструктора
  powers() { x = 0; } // no initializer
  powers(int n) { x = n; } // initializer

  int getx() { return x; }
  void setx(int i) { x = i; }
};

int main()
{
  powers ofTwo[] = {1, 2, 4, 8, 16}; // Инициализированный массив
  powers ofThree[5]; // Неинициализированный массив
  powers *p;
  int i;

  // Степени двойки
  cout << "Степени двойки: ";
  for(i=0; i<5; i++) {
    cout << ofTwo[i].getx() << " ";
  }
  cout << "\n\n";

  // Степени тройки
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
```

```
  ofThree[4].setx(81);

  // Выводим степени тройки
  cout << "Степени тройки: ";
  for(i=0; i<5; i++) {
    cout << ofThree[i].getx() << " ";
  }
  cout << "\n\n";

  // Динамический массив
  try {
    p = new powers[5]; // Без инициализации
  } catch (bad_alloc xa) {
      cout << "исключительная ситуация\n";
      return 1;
  }

  // Инициализация динамического массива степенями двойки
    for(i=0; i<5; i++) {
    p[i].setx(ofTwo[i].getx());
  }

  // Выводим степени двойки
  cout << "Степени двойки: ";
  for(i=0; i<5; i++) {
    cout << p[i].getx() << " ";
  }
  cout << "\n\n";

  delete [] p;
  return 0;
}
```

В этом примере оба конструктора необходимы. Конструктор по умолчанию используется для создания неинициализированного массива **ofThree** и динамического массива, а конструктор с параметрами вызывается для создания объектов, хранящихся в массиве **ofTwo**.

Конструктор копирования

Одним из важнейших видов перегруженного конструктора является *конструктор копирования* (copy constructor). Он позволяет предотвратить проблемы, которые могут возникнуть при присваивании одного объекта другому.

Рассмотрим проблему, для решения которой необходим конструктор копирования. По умолчанию, если один объект инициализируется другим, создается побитовая копия присваиваемого объекта. Иначе говоря, инициализируемому объекту присваивается его идентичная копия. Иногда побитовая копия оказывается неприемлемой. Обычно это происходит, когда объект выделяет динамическую память. Например, допустим, что класс **MyClass** выделяет динамическую память для каждого создаваемого объекта, а объект **A** является его экземпляром. Это значит, что объект **A** уже выделил для себя динамическую память. Теперь предположим, что объект **A** инициализирует объект **B**, как показано ниже.

```
MyClass B = A;
```

Если при этом создается побитовая копия объекта **A**, то объект **B** будет точной копией объекта **A**. Следовательно, объект **B** также будет ссылаться на область памяти, выделенную объектом **A**, не выделяя свой собственный участок. Очевидно, это совсем не то, к чему мы стремились. Например, если класс **MyClass** содержит деструктор, освобождающий память, то при уничтожении объектов **A** и **B** одна и та же область памяти будет освобождаться дважды!

Эта проблема может возникнуть еще в двух ситуациях. Во-первых, когда функции передается копия параметра, и, во-вторых, когда создается временный объект, возвращаемый функцией. Напомним, что функция создает временный объект, в котором хранится возвращаемое ею значение.

Для решения этих проблем предназначен конструктор копирования, который не использует побитовое копирование. Наиболее часто применяется следующая его форма.

```
имя_класса(const имя_класса &ссылка_не_объект) {
   // Тело конструктора
}
```

Здесь *ссылка на объект* связана с объектом, стоящим в правой части инициализации. Конструктор копирования может иметь дополнительные параметры. Однако в любом случае первым параметром должна быть ссылка на инициализирующий объект.

Следует помнить, что существуют две ситуации, в которых один объект может присваиваться другому. Во-первых, при выполнении оператора присваивания. Во-вторых, при инициализации, которую можно осуществить тремя способами.

- Явная инициализация, например, при объявлении объекта.
- Создание копии объекта, передаваемого функции в качестве параметра.
- Создание временного объекта (чаще всего, при возврате значения функции).

Конструктор копирования применяется только для инициализации. Рассмотрим класс **myclass** и его объект **y**. Инициализация будет происходить при выполнении каждого из этих операторов.

```
myclass x = y; // Объект y явно инициализируется объектом x
func(y);       // Объект y передается как параметр
y = func();    // Объекту y присваивается возвращаемый объект
```

Рассмотрим программу, в которой необходим явный конструктор копирования. Эта программа создает очень ограниченный и "безопасный" тип целочисленного массива, предотвращающий выход индекса за пределы допустимого диапазона. (В главе 15 описан более эффективный способ решения этой проблемы, основанный на перегрузке операторов.) С помощью оператора **new** каждому элементу массива выделяется динамическая память. Причем указатель на выделенную область памяти хранится внутри каждого объекта.

```
/* Программа создает "безопасный" тип массива.
   Поскольку память для каждого элемента массива выделяется
   с помощью оператора new, для их инициализации применяется
   конструктор копирования.
*/
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
  int *p;
  int size;
```

```
public:
  array(int sz) {
    try {
      p = new int[sz];
    } catch (bad_alloc xa) {
      cout << "Исключительная ситуация\n";
      exit(EXIT_FAILURE);
    }
    size = sz;
  }
  ~array() { delete [] p; }

  // Конструктор копирования
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i] = j;
  }

  int get(int i) {
    return p[i];
  }
};

// Конструктор копирования
array::array(const array &a) {
  int i;

  try {
    p = new int[a.size];
  } catch (bad_alloc xa) {
    cout << "Исключительная ситуация\n";
    exit(EXIT_FAILURE);
  }
  for(i=0; i<a.size; i++) p[i] = a.p[i];
}

int main()
{
  array num(10);
  int i;

  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // Создается другой массив, который инициализируется
  // массивом num
  array x(num); // Вызов конструктора копирования
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

Рассмотрим, что произойдет, когда массив **num** инициализирует массив **x** с помощью оператора

```
array x(num); // Вызов конструктора копирования
```

При вызове конструктора копирования для нового массива выделяется область динамической памяти. Указатель на ее первую ячейку хранится в указателе **x.p**. Таким образом, объекты **x** и **num** содержат одинаковые элементы, однако каждый из них занимает отдельную область памяти. Если бы в программе не использовался конструктор копирования, объекты **x** и **num** занимали бы одну область динамической памяти. (Это значит, что указатели **num.p** и **x.p** содержали бы один и тот же адрес.)

Напомним, что конструктор копирования вызывается только для инициализации. Например, следующие операторы не вызывают конструктор копирования.

```
array a(10);
// ...
array b(10);

b = a; // Конструктор копирования не вызывается
```

В данном случае оператор **b=a** выполняет присваивание одного объекта другому. Если оператор присваивания не перегружен (как в данном случае), создается побитовая копия объекта, расположенного в его правой части. Следовательно, в некоторых ситуациях возникает необходимость не только предусмотреть конструктор копирования, но и перегрузить оператор присваивания (см. главу 15).

Определение адреса перегруженой функции

Как указано в главе 5, функция имеет адрес. Этот адрес можно присвоить указателю, а затем вызывать функцию не по имени, а через ее указатель. Если функция не перегружена, этот процесс очень прост. Однако для перегруженных функций ситуация немного сложнее. Чтобы понять причины этого, рассмотрим оператор, в котором указателю **p** присваивается адрес функции **myfunc()**.

```
p = myfunc;
```

Если функция **myfunc()** не перегружена, она существует в одном экземпляре, и компилятор без труда вычисляет ее адрес. Однако, если функция **myfunc()** перегружена, возникает вопрос, каким образом компилятор может вычислить ее указатель? Ответ зависит от того, как объявлен указатель **p**. Рассмотрим, например, следующую программу.

```
#include <iostream>
using namespace std;

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  int (*fp)(int a); // Указатель на функцию int f(int)

  fp = myfunc; // Указатель на функцию myfunc(int)

  cout << fp(5);

  return 0;
}

int myfunc(int a)
{
```

```
  return a;
}

int myfunc(int a, int b)
{
  return a*b;
}
```

В этой программе существуют два варианта функции **myfunc()**. Оба возвращают целое число, однако первый вариант получает один целочисленный аргумент, а второй — два. Переменная **fp** объявлена в программе как указатель на функцию, возвращающую целое значение и получающую один целочисленный аргумент. Когда указателю **fp** присваивается адрес функции **myfunc()**, компилятор использует эту информацию для выбора соответствующей версии, а именно: **myfunc(int a)**. Если бы указатель **fp** был объявлен иначе:

```
int (*fp)(int a, int b);,
```

то ему был бы присвоен адрес функции **myfunc(int a, int b)**.

Как правило, если указателю присваивается адрес перегруженной функции, выбор варианта осуществляется в зависимости от типа указателя. Кроме того, объявление указателя на функцию должно точно соответствовать только одному из вариантов перегруженной функции.

Анахронизм overload

В первоначальной версии языка C++ для создания перегруженной функции требовалось ключевое слово **overload**. В настоящее время оно устарело и больше не поддерживается. Действительно, его нет даже в списке ключевых слов языка C++. Однако, поскольку мы допускаем существование старых программ, представляющих как практический, так и исторический интерес, неплохо вспомнить, как применялось ключевое слово **overload**. Рассмотрим общую форму такого объявления.

```
overload имя_функции;
```

Здесь имя функции задает перегружаемую функцию. Этот оператор должен предшествовать объявлению перегруженной функции. Например, следующий оператор сообщает старым компиляторам, что функция **test()** является перегруженной.

```
overload test;
```

Аргументы функции по умолчанию

В языке C++ аргументам функции можно присваивать значения, заданные по умолчанию, если соответствующий аргумент при вызове функции был пропущен. Значение по умолчанию задается с помощью синтаксической конструкции, которая очень похожа на инициализацию переменной. Например, следующий оператор объявляет, что функция **myfunc()** получает один аргумент типа **double**, по умолчанию принимающий значение 0.0.

```
void myfunc(double d = 0.0)
{
  // ...
}
```

Теперь функцию **myfunc()** можно вызвать двумя способами.

```
myfunc(198.234); // Передача явного значения
myfunc();        // Функция использует значение по умолчанию
```

При первом вызове параметр **d** получает значение 198.234. Во время второго вызова параметр **d** автоматически принимает значение 0.0.

Параметры по умолчанию позволяют справляться с возрастающей сложностью программ. Во многих случаях функции содержат больше параметров, чем нужно в конкретной ситуации. Таким образом, в каждом конкретном случае достаточно указать лишь необходимые параметры, а не все сразу. Например, многие функции ввода-вывода используют именно этот механизм.

Рассмотрим простую программу, иллюстрирующую применение параметров функции по умолчанию. Функция **clrscr()** очищает экран дисплея, стирая все строки (кстати, это не самый эффективный способ). Поскольку, как правило, в текстовом режиме на экране дисплея помещаются 25 строк, параметр функции по умолчанию принимает значение 25. Однако в некоторых случаях на экран может выводиться разное количество строк, как больше, так и меньше 25. Поэтому параметр по умолчанию можно заменить явным значением.

```
#include <iostream>
using namespace std;

void clrscr(int size=25);

int main()
{
  register int i;

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(); // Стирает 25 строк

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(10); // Стирает 10 строк

  return 0;
}

void clrscr(int size)
{
   for(; size; size--) cout << endl;
}
```

Этот пример иллюстрирует ситуацию, в которой достаточно параметра по умолчанию. В этом случае функция **clrscr()** вызывается без параметров. Однако при необходимости значение по умолчанию можно заменить и с помощью переменной **size** явно задать другое значение.

Аргумент по умолчанию можно применять в качестве признака, сообщающего функции, что ей необходимо использовать предыдущее значение аргумента. Проиллюстрируем эту ситуацию с помощью функции **iputs()**, предназначенной для автоматической вставки перед строкой заданного количества пробелов. Для начала рассмотрим версию этой функции, в которой не предусмотрен аргумент по умолчанию.

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent = 0;
```

```
  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

Эта версия функции **iputs()** вызывается с двумя аргументами: строкой и количеством пробелов перед ней. Однако функцию **inputs()** можно усовершенствовать, предусмотрев для аргумента **indent** значение по умолчанию. Очень часто на экран выводятся строки, имеющие определенный отступ от края экрана. В этой ситуации можно не повторять аргумент **indent** постоянно, а просто задать его значение по умолчанию и вызвать функцию **iputs()** с одним параметром **str**. Этот подход иллюстрируется следующей программой.

```
#include <iostream>
using namespace std;

/* По умолчанию параметр indent равен -1.  Это значение
   заставляет функцию использовать предыдущую величину
   отступа. */
void iputs(char *str, int indent = -1);

int main()
{
  iputs("Общий привет", 10);
  iputs("Перед этой строкой вставлены 10 пробелов");
  iputs("Перед этой строкой вставлены 5 пробелов", 5);
  iputs("Перед этой строкой нет пробелов", 0);

  return 0;
}

void iputs(char *str, int indent)
{
  static i = 0; // Применяется предыдущее значение параметра indent

  if(indent >= 0)
    i = indent;
  else  // Используется старое значение параметра indent
    indent = i;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

Эта программа выводит на экран следующие строки.

```
          Всем привет
          Перед этой строкой вставлены 10 пробелов
     Перед этой строкой вставлены 5 пробелов
Перед этой строкой нет пробелов
```

Создавая функции с аргументами по умолчанию, важно помнить, что их значения можно задать лишь один раз во время объявления функции. В предыдущем примере значение по умолчанию было указано в прототипе функции **iputs()**. Если попытаться задать новое (или даже то же самое значение) в определении функции **iputs()**, компилятор выдаст сообщение об ошибке. Хотя значения аргументов по умолчанию

переопределить невозможно, можно задавать разные значения для каждой версии перегруженной функции.

Все параметры, принимающие значения по умолчанию, должны располагаться правее обычных аргументов. Например, следующее определение функции **iputs()** является неправильным.

```
// Ошибка!
void iputs(int indent = -1, char *str);
```

Начав определять параметры, принимающие значения по умолчанию, нельзя перемежать их обычными параметрами. Иначе говоря, следующее объявление является неверным.

```
int myfunc(float f, char *str, int i=10, int j);
```

Поскольку значение параметра **i** задается по умолчанию, параметр **j** также должен иметь значение по умолчанию.

Параметры конструктора тоже могут иметь значения по умолчанию. Например, в классе **cube** задаются целочисленные размеры куба. Если конструктор вызывается без параметров, размеры куба по умолчанию равны нулю.

```
#include <iostream>
using namespace std;

class cube {
  int x, y, z;
public:
  cube(int i=0, int j=0, int k=0) {
    x=i;
    y=j;
    z=k;
  }

  int volume()
  {
    return x*y*z;
  }
};

int main()
{
  cube a(2,3,4), b;

  cout << a.volume() << endl;
  cout << b.volume();

  return 0;
}
```

Параметры конструктора, заданные по умолчанию, предоставляют два преимущества. Во-первых, они позволяют избежать применения перегруженного конструктора без параметров. Например, если бы параметры конструктора **cube()** не были заданы по умолчанию, для объявления объекта **b** понадобился бы второй конструктор:

```
cube() {x=0; y=0; z=0}
```

Во-вторых, задать самые распространенные значения аргументов по умолчанию удобнее, чем повторять их постоянно.

Аргументы по умолчанию и перегрузка

В некоторых ситуациях аргументы по умолчанию оказываются сокращенной формой перегрузки. Примером этой ситуации является класс **cube**. Теперь рассмотрим еще один. Допустим, необходимо создать две настраиваемые версии стандартной функции **strcat()**. Первая версия будет практически совпадать с функцией **strcat()** и конкатенировать целые строки. Вторая версия предусматривает третий параметр, задающий количество конкатенируемых символов. Иначе говоря, вторая версия этой функции приписывает к первой строке лишь заданное количество символов из второй строки. Итак, функция **mystrcat()** будет иметь следующие прототипы.

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

Первая версия функции **mystrcat()** дописывает **len** символов строки **s2** в конец строки **s1**. Вторая версия дописывает в конец строки **s1** всю строку **s2**.

Разумеется, можно было бы реализовать обе версии функции **mystrcat()**, однако есть более простой способ. Используя аргумент по умолчанию, можно создать лишь одну версию функции **mystrcat()** для обоих вариантов. Этот способ продемонстрирован с помощью следующей программы.

```
// Настраиваемая версия функции strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);

int main()
{
  char str1[80] = "Проверка";
  char str2[80] = "0123456789";

  mystrcat(str1, str2, 5); // Конкатенируем 5 символов
  cout << str1 << '\n';

  strcpy(str1, "Проверка"); // Строка str1

  mystrcat(str1, str2); // Конкатенируем целые строки
  cout << str1 << '\n';

  return 0;
}

// Настраиваемая версия функции strcat().
void mystrcat(char *s1, char *s2, int len)
{
  // Находим конец строки s1
  while(*s1) s1++;

  if(len == -1) len = strlen(s2);

  while(*s2 && len) {
    *s1 = *s2; // Копируем символы
    s1++;
    s2++;
    len--;
  }
```

```
  *s1 = '\0'; // Признак конца строки s1
}
```

Здесь функция **mystrcat()** конкатенирует **len** символов из строки, на которую ссылается указатель **s2**, в конец строки, на которую ссылается указатель **s1**. Однако, если параметр **len** по умолчанию равен −1, функция **mystrcat()** конкатенирует целые строки. (Таким образом, если параметр **len** равен −1, функция **mystrcat()** работает как стандартная функция **strcat()**.) Используя параметр по умолчанию, можно объединить две версии функции в одной. Иначе говоря, в этом смысле параметры по умолчанию являются альтернативой перегрузке функций.

Правильное применение аргументов по умолчанию

Аргументы по умолчанию представляют собой мощный механизм, который, однако, зачастую используется неверно. Основное предназначение аргументов по умолчанию — обеспечить простой, естественный и эффективный стиль программирования, сохранив высокую гибкость программ. Следовательно, аргументы по умолчанию следует применять для очень распространенных ситуаций, в которых вызываются функции. Если аргумент не имеет определенного значения, которое он принимает в большинстве ситуаций, то его значение по умолчанию не стоит предусматривать вообще. Если значение по умолчанию выбрано непродуманно, оно лишь запутывает код и может привести к различным недоразумениям.

Кроме того, значение аргумента по умолчанию не должно вызывать опасных последствий для программы. Иначе говоря, непредвиденное применение значения по умолчанию не должно приводить к краху программы.

Перегрузка функций и неоднозначность

Можно создать ситуацию, в которой компилятор не сможет сделать выбор среди нескольких перегруженных функций. Такие ситуации называются *неоднозначными* (ambiguous). Они являются ошибками, и программы, содержащие их, не компилируются.

Основной причиной неоднозначности в программах на языке C++ является автоматическое преобразование типов. Как известно, компилятор пытается автоматически преобразовать тип фактических параметров к типу формальных аргументов, ожидаемых функцией. Рассмотрим следующий фрагмент программы.

```
int myfunc(double d);
// ...
cout << myfunc('c'); // Это не ошибка, преобразование допускается
```

Как указано в комментариях, этот фрагмент не содержит синтаксической ошибки, поскольку символ **c** можно преобразовать в тип **double**. Набор запрещенных преобразований типов в языке C++ довольно мал. И все же, несмотря на удобство, которое обеспечивает автоматическое преобразование типов, оно часто порождает неоднозначные ситуации. Рассмотрим в качестве примера следующую программу.

```
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);
```

```
int main()
{
  cout << myfunc(10.1) << " "; // Однозначная ситуация,
                               // вызывается функция
                               // myfunc(double)
  cout << myfunc(10); // Неоднозначность

  return 0;
}

float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

В этом примере функция **myfunc()** перегружена, поскольку она может получать аргумент типа **float** или **double**. В однозначной ситуации функция **myfunc(double)** вызывается без проблем, поскольку все аргументы типа **float** автоматически преобразовываются в тип **double**. Это совершенно однозначное преобразование. В то же время вызов **myfunc(10)** порождает неоднозначную ситуацию, поскольку компилятор не знает, в какой тип следует преобразовать целый аргумент — **float** или **double**. В таком случае на экране появится сообщение об ошибке, и компиляция будет прекращена.

Как следует из этого примера, неоднозначность порождается вовсе не перегрузкой функции **myfunc()**. Ее причиной является конкретный вызов функции **myfunc()**, в котором используется неопределенный тип аргумента.

Рассмотрим еще один пример неоднозначной ситуации, порождаемой автоматическим преобразованием типов.

```
#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
  cout << myfunc('c'); // Вызов функции myfunc(char)
  cout << myfunc(88) << " "; // Неоднозначность

  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
}

char myfunc(char ch)
{
  return ch+1;
}
```

В языке C++ типы **unsigned char** и **char** сами по себе не порождают неоднозначности. Однако, когда выполняется вызов **myfunc(88)**, компилятор не знает, какой вариант функции вызывать. Иначе говоря, совершенно непонятно, в какой тип следует преобразовать аргумент — в **unsigned char** или **char**.

Неоднозначные ситуации могут возникнуть также из-за аргументов перегруженных функций, заданных по умолчанию. Изучим следующую программу.

```
#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
  cout << myfunc(4, 5) << " "; // Однозначная ситуация
  cout << myfunc(10); // Неоднозначная ситуация

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

Здесь в первом вызове функции **myfunc()** заданы два аргумента. Следовательно, никакой неоднозначности нет, и вызывается функция **myfunc(int i, int j)**. Однако при втором вызове функции **myfunc()** возникает неоднозначность, поскольку компилятор не знает, какой вариант вызывать: то ли версию с одним аргументом, то ли версию с двумя аргументами, второй из которых задан по умолчанию.

Некоторые виды перегруженных функций неоднозначны по своей природе, несмотря на то что, на первый взгляд, они выглядят абсолютно точными. Рассмотрим в качестве примера следующую программу.

```
// В этой программе есть ошибка.
#include <iostream>
using namespace std;

void f(int x);
void f(int &x); // Ошибка

int main()
{
  int a=10;

  f(a); // Ошибка, какую версию функции f() следует вызывать?

  return 0;
}
```

```
void f(int x)
{
  cout << "Внутри функции f(int)\n";
}

void f(int &x)
{
  cout << "Внутри функции f(int &)\n";
}
```

Как указано в комментариях программы, две функции нельзя перегружать, если вся разница между ними заключается в том, что одна из них имеет параметр, передаваемый с помощью ссылки, а вторая получает аргумент по значению. В этой ситуации компилятор не сможет выбрать однозначный вариант. Следует помнить, что между передачей параметров по ссылке и по значению нет синтаксической разницы.

Полный справочник по

Глава 15

Перегрузка операторов

С перегрузкой функций тесно связан механизм перегрузки операторов. В языке C++ можно перегрузить большинство операторов, настроив их на конкретный класс. Например, в классе, поддерживающем стек, оператор "+" можно перегрузить для заталкивания элементов в стек, а оператор "-" — для выталкивания элементов из него. Перегруженный оператор сохраняет свое первоначальное предназначение. Просто набор типов, к которым его можно применять, расширятся.

Перегрузка операторов — одна из самых эффективных возможностей языка C++. Она позволяет полностью интегрировать новые классы в существующую программную среду. После перегрузки операции над объектами новых классов выглядят точно так же, как операции над переменными встроенных типов. Кроме того, перегрузка операторов лежит в основе системы ввода-вывода в языке C++.

Перегрузка операторов осуществляется с помощью *операторных функций* (operator function), которые определяют действия перегруженных операторов применительно к соответствующему классу. Операторные функции создаются с помощью ключевого слова **operator**. Операторные функции могут быть как членами класса, так и обычными функциями. Однако обычные операторные функции, как правило, объявляют дружественными по отношению к классу, для которого они перегружают оператор. В каждом из этих случаев операторная функция объявляется по-разному. Следовательно, необходимо рассмотреть эти варианты отдельно.

Создание операторной функции-члена

Операторная функция-член имеет следующий вид:

```
тип_возвращаемого_значения имя_класса::operator#(список-аргументов)
{
  ... // Операции
}
```

Обычно операторная функция возвращает объект класса, с которым она работает, однако тип возвращаемого значения может быть любым. Символ # заменяется перегружаемым оператором. Например, если в классе перегружается оператор деления "/", операторная функция-член называется **operator/**. При перегрузке унарного оператора *список аргументов* остается пустым. При перегрузке бинарного оператора *список аргументов* содержит один параметр. (Эти необычные правила мы разъясним позже.)

Рассмотрим простой пример. Эта программа создает класс **loc**, в котором хранятся географические координаты: широта и долгота. В ней перегружается оператор "+". Внимательно изучите эту программу, уделяя особое внимание определению функции **operator+()**.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }
```

```
  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1.show(); // Выводит на экран числа 10 20
  ob2.show(); // Выводит на экран числа 5 30

  ob1 = ob1 + ob2;
  ob1.show(); // Выводит на экран числа 15 50

  return 0;
}
```

Как видим, функция **operator+()** имеет только один параметр, несмотря на то, что она перегружает бинарный оператор. (Как известно, бинарные операторы имеют два операнда.) Причина заключается в том, что операнд, стоящий в левой части оператора, передается операторной функции неявно с помощью указателя **this**. Операнд, стоящий в правой части оператора, передается операторной функции через параметр **op2**. Отсюда следует важный вывод: при перегрузке бинарного оператора вызов операторной функции генерируется объектом, стоящим в левой части оператора.

Как правило, перегруженные операторные функции возвращают объект класса, с которым они работают. Следовательно, перегруженный оператор можно использовать внутри выражений. Например, если бы операторная функция **operator+()** возвращала объект другого типа, следующее выражение было бы неверным.

```
ob1 = ob1 + ob2;
```

Для того чтобы присвоить сумму объектов **ob1** и **ob2** объекту **ob1**, необходимо, чтобы результат операции **+** имел тип **loc**.

Кроме того, поскольку операторная функция **operator+()** возвращает объект типа **loc**, допускается следующее выражение:

```
(ob1+ob2).show(); // Выводит на экран сумму ob1+ob2
```

В этой ситуации операторная функция создает временный объект, который уничтожается после возвращения из функции **show()**.

Следует понимать, что операторная функция может возвращать объекты любых типов, и что эти типы зависят только от конкретного приложения. Однако, *как правило*, операторные функции возвращают объекты классов, с которыми они работает.

И еще одно замечание: операторная функция **operator+()** не модифицирует свои операнды. Поскольку традиционный оператор "+" не изменяет свои операнды, не имеет смысла предусматривать для функции **operator+()** иное поведение. (Например, 5+7 равно 12, но при этом слагаемые по-прежнему равны 5 и 7.) Несмотря на то что содержание операторной функции можно определять произвольно, следует оставаться в рамках здравого смысла.

Следующая программа дополняет класс **loc** тремя перегруженными операторами: "-", "=" и унарным оператором "++". Обратите особое внимание на объявления этих функций.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {} // Этот конструктор необходим для создания
           // временных объектов
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Перегруженный оператор + для класса loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

// Перегруженный оператор - для класса loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // Обратите внимание на порядок операндов
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}
```

```
// Перегруженный оператор присваивания для класса loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // Возвращает объект, генерирующий вызов
}

// Перегруженный префиксный оператор инкрементации ++
// для класса loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);

  ob1.show();
  ob2.show();

  ++ob1;
  ob1.show(); // Выводит на экран числа 11 21

  ob2 = ++ob1;
  ob1.show(); // Выводит на экран числа 12 22
  ob2.show(); // Выводит на экран числа 12 22

  ob1 = ob2 = ob3; // Множественное присваивание
  ob1.show(); // Выводит на экран числа 90 90
  ob2.show(); // Выводит на экран числа 90 90

  return 0;
}
```

Рассмотрим, например, операторную функцию **operator-()**. Обратите внимание на порядок ее операндов. Учитывая смысл оператора вычитания, операнд, стоящий в его правой части, вычитается из операнда, стоящего слева. Поскольку вызов операторной функции **operator-()** генерируется объектом, стоящим слева от знака "минус", данные объекта **op2** должны вычитаться из данных объекта, на который ссылается указатель **this**. Всегда следует помнить, какой объект генерирует вызов операторной функции.

Если оператор "=" не перегружен, для класса автоматически создается оператор присваивания по умолчанию, который создает побитовые копии объектов. Перегружая оператор "=", можно явно определить оператор присваивания для конкретного класса. В данном примере оператор присваивания ничем не отличается от стандартного, однако в других ситуациях он может выполнять иные действия. Обратите внимание на то, что функция **operator=()** возвращает указатель ***this** на объект, сгенерировавший ее вызов. Это позволяет выполнять множественное присваивание объектов, как показано ниже.

```
ob1 = ob2 = ob3; // Множественное присваивание
```

Рассмотрим теперь определение операторной функции **operator++()**. Как видим, эта функция не имеет параметров. Поскольку оператор инкрементации "**++**" является унарным, его единственный операнд неявно передается ему с помощью указателя **this**.

Обратите внимание на то, что операторные функции **operator=()** и **operator++()** изменяют значения своих операндов. Например, левому операнду функции **operator=()** (генерирующему вызов функции) присваивается новое значение. Кроме того, оператор инкрементации увеличивает значение своего операнда на единицу. Как указывалось ранее, хотя содержание оператора ничем не регламентируется, всегда следует придерживаться его первоначального смысла.

Создание префиксной и постфиксной форм операторов инкрементации и декрементации

В предыдущей программе был перегружен только оператор инкрементации в префиксной форме. Однако стандарт языка C++ позволяет явно создавать отдельные версии префиксного и постфиксного операторов инкрементации и декрементации. Для этого следует определить две версии функции **operator++()** (и функции **operator--()** соответственно). Одна из этих версий была определена в предыдущей программе, а другую мы определим ниже.

```
loc operator++(int x);
```

Если символы **++** предшествуют операндам, вызывается операторная функция **operator++()**, если символы **++** следуют за операндами, вызывается операторная функция **operator++(int x)**.

Предыдущий пример можно обобщить. Вот как выглядит общая форма префиксной и постфиксной операторных функций **operator++()** и **operator--()**.

```
// Префиксный оператор инкрементации
тип operator++() {
    // Тело префиксного оператора
}

// Постфиксный оператор инкрементации
тип operator++(int i) {
    // Тело постфиксного оператора
}

// Префиксный оператор декрементации
тип operator--() {
    // Тело префиксного оператора
}

// Постфиксный оператор декрементации
тип operator--(int i) {
    // Тело постфиксного оператора
}
```

На заметку

Работая со старыми программами на языке C++, содержащими операторы инкрементации и декрементации, следует проявлять осторожность. В этих программах невозможно различить префиксную и постфиксную форму перегруженных операторов "++" и "--". В обоих случаях следует применять префиксную форму.

Перегрузка сокращенных операторов присваивания

Сокращенные операторы присваивания (например, "+=", "-=" и т.п.) также можно перегружать. Рассмотрим операторную функцию **operator+=()** для класса **loc**.

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;

  return *this;
}
```

Ограничения на перегруженные операторы

На применение перегруженных операторов налагается несколько ограничений. Во-первых, нельзя изменить приоритет оператора. Во-вторых, невозможно изменить количество операндов оператора. (Однако операнд можно игнорировать.) В-третьих, операторную функцию нельзя вызывать с аргументами, значения которых заданы по умолчанию (за исключением оператора вызова функции, который мы рассмотрим немного позднее). И, в заключение, нельзя перегружать следующие операторы:

```
.   ::   .*   ?
```

Как неоднократно указывалось, внутри операторной функции можно программировать любые операции. Например, никто не запрещает перегружать оператор "+" так, чтобы он десять раз записывал в файл строку "Я люблю C++". Однако эти действия сильно отличаются от обычного смысла оператора "+", и вы рискуете разрушить свою программу. Если кто-нибудь станет читать ее и увидит оператор **Ob1+Ob2**, то, естественно, он предположит, что в этом месте складываются два объекта, а не выполняется операция вывода в файл. Следовательно, для значительного отклонения от обычного предназначения оператора должны существовать веские причины. Одна из них — необходимость перегружать операторы "<<" и ">>". Хотя операторы побитового сдвига не имеют никакого отношения к вводу-выводу, их внешний вид настолько хорошо соответствует этому предназначению, что невозможно устоять перед искушением перегрузить их новым смыслом. И все же, как правило, лучше оставаться в рамках традиционного смысла.

За исключением оператора "=", операторные функции наследуются производными классами. Однако в производном классе каждый из этих операторов снова можно перегрузить.

Перегрузка операторов с помощью дружественных функций

Операторы можно перегружать с помощью дружественных функций, не являющихся членами класса. Это значит, что дружественные функции не получают неявного указателя **this**. Следовательно, перегруженная операторная функция получает параметры явно. Таким образом, при перегрузке бинарного оператора дружественная функция получает два параметра, а при перегрузке унарного оператора — один. Первым параметром дружественной функции, перегружающей бинарный оператор, является его левый операнд, а вторым — правый операнд.

В следующей программе функция **operator+()** является дружественной по отношению к классу **loc**.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {} // Этот конструктор необходим для создания
           // временных объектов
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, loc op2); // Дружественная функция
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Оператор + перегружается с помощью дружественной функции.
loc operator+(loc op1, loc op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;

  return temp;
}

// Перегруженный оператор - для класса loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // Обратите внимание на порядок операндов
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Перегрузка оператора присваивания для класса loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // Возвращается объект, генерирующий вызов
}
```

```
// Перегруженный оператор ++ для класса loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1 = ob1 + ob2;
  ob1.show();

  return 0;
}
```

На применение дружественных операторных функций налагаются несколько ограничений. Во-первых, с помощью дружественных функций нельзя перегружать операторы "=", "()", "[]" и "->". Во-вторых, как будет показано в следующем разделе, при перегрузке операторов инкрементации и декрементации параметр дружественной функции следует передавать по ссылке.

Применение дружественных функций для перегрузки операторов "++" и "--"

Если дружественные функции применяются для перегрузки операторов инкрементации и декрементации, их операнды следует передавать с помощью ссылок. Это необходимо потому, что дружественная функция не получает указателя **this**. Предположим, что мы сохраняем первоначальный смысл операторов "++" и "--". Следовательно, оператор "++", например, должен модифицировать свой операнд. Если перегрузить этот оператор с помощью дружественной функции, как обычно, то операнд будет передаваться по значению. Это значит, что дружественная функция не сможет изменить свой параметр. Однако эту проблему можно решить, если передать операнд дружественной функции с помощью ссылки. В этом случае все изменения внутри функции будут отражаться на ее фактическом параметре. Например, следующая программа использует дружественные функции для перегрузки операторов "++" и "--" в классе **loc**.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator=(loc op2);
```

```
  friend loc operator++(loc &op);
  friend loc operator--(loc &op);
};

// Перегруженный оператор присваивания для класса loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // Возвращается объект, генерирующий вызов
}

// Оператор ++ перегружен дружественной функцией;
// используется передача параметра с помощью ссылки.
loc operator++(loc &op)
{
  op.longitude++;
  op.latitude++;

  return op;
}

// Оператор ++ перегружен дружественной функцией;
// используется передача параметра с помощью ссылки.
loc operator--(loc &op)
{
  op.longitude--;
  op.latitude--;

  return op;
}

int main()
{
  loc ob1(10, 20), ob2;

  ob1.show();
  ++ob1;
  ob1.show(); // Выводит на экран числа 11 21

  ob2 =  ++ob1;
  ob2.show(); // Выводит на экран числа 12 22

  --ob2;
  ob2.show(); // Выводит на экран числа 11 21

  return 0;
}
```

Если необходимо перегрузить постфиксную форму оператора инкрементации или декрементации, следует просто задать второй, фиктивный целочисленный параметр, как показано ниже.

```
// Перегрузка постфиксной формы оператора ++
// с помощью дружественной функции
friend loc operator++(loc &op, int x);
```

Дружественные операторные функции повышают гибкость

Во многих случаях способ перегрузки операторов не важен: функции-члены и дружественные функции эквивалентны. В таких ситуациях следует предпочесть функции-члены. Однако бывают ситуации, в которых дружественные операторные функции позволяют повысить гибкость перегруженного оператора.

Как известно, при перегрузке бинарного оператора с помощью функции-члена вызов функции осуществляется объектом, расположенным в левой части оператора. Более того, функция-член получает указатель **this** на этот объект. Теперь допустим, что в некоем классе операторная функция-член **operator+()** прибавляет к объекту целое число. Назовем объект этого класса именем **Ob**. В таком случае следующее выражение является вполне допустимым.

```
Ob + 100; // Все правильно
```

Здесь объект **Ob** вызывает функцию **operator+()**, выполняющую сложение. А что произойдет, если это выражение переписать иначе?

```
100 + Ob; // Неправильно
```

Теперь в левой части оператора стоит целое число. Поскольку оно имеет встроенный тип, операция сложения целого числа и объекта **Ob** не определена. Следовательно, компилятор сочтет это выражение ошибочным. Можно себе представить, насколько усложнится программирование, если вы будете вынуждены постоянно следить за положением объектов в выражениях.

Эту проблему легко решить с помощью дружественной операторной функции. В этом случае дружественная функция получит оба параметра. Следовательно, выражения вида *объект+целое число* и *целое число+объект* станут правильными, поскольку дружественной функции не важен порядок следования операндов.

Проиллюстрируем сказанное следующей программой.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};

// Оператор + перегружен для сложения объекта с целым числом.
loc operator+(loc op1, int op2)
{
  loc temp;
```

```
  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;

  return temp;
}
// Оператор + перегружен для сложения целого числа с объектом.
loc operator+(int op1, loc op2)
{
  loc temp;

  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

  ob1.show();
  ob2.show();
  ob3.show();

  ob1 = ob2 + 10; // Оба выражения
  ob3 = 10 + ob2; // верны

  ob1.show();
  ob3.show();

  return 0;
}
```

Перегрузка операторов new и delete

В языке C++ можно перегружать операторы **new** и **delete**. Это приходится делать, если возникает необходимость создать особый механизм распределения памяти. Например, можно потребовать, чтобы процедура распределения памяти использовала жесткий диск в качестве виртуальной памяти, если куча исчерпана. Перегрузка операторов **new** и **delete** осуществляется очень просто.

Рассмотрим схематическое устройство функций, перегружающих операторы **new** и **delete**.

```
// Выделение памяти для объекта.
void *operator new(size_t size)
{
  /* Выделяется память. В случае неудачи генерируется
     исключительная ситуация bad_alloc.
     Конструктор вызывается автоматически. */
  return pointer_to_memory;
}
```

```
// Удаление объекта.
void operator delete(void *p)
{
  /* Освобождается память, на которую ссылается указатель p.
     Деструктор вызывается автоматически. */
}
```

Размер типа **size_t** соответствует единице выделяемой памяти. (Как правило, тип **size_t** определяется как тип **unsigned int**.) Параметр **size** задает количество байтов, необходимых для хранения объекта, размещаемого в памяти. Перегруженная функция **new** должна возвращать указатель на выделенную область памяти или генерировать исключительную ситуацию **bad_alloc**. Во всем остальном функция **new** может быть абсолютно произвольной. При выделении памяти с помощью оператора **new** (как стандартного, так и перегруженного) автоматически вызывается конструктор объекта.

Оператор **delete** получает указатель на область освобождаемой памяти и возвращает ее операционной системе. Для уничтожения объекта автоматически вызывается деструктор.

Для того чтобы операторы **new** и **delete** можно было вызывать в любых местах программы, их следует перегружать глобально. Кроме того, их можно перегружать лишь для отдельных классов. Рассмотрим перегрузку операторов **new** и **delete** для конкретного класса. Для простоты будем предполагать, что память выделяется как обычно, т.е. с помощью стандартных функций **malloc()** и **calloc()**. (Разумеется, вы можете реализовать свои собственные схемы распределения памяти.)

Для перегрузки операторов **new** и **delete** в классе следует определить перегруженные операторные функции-члены. Например, в следующей программе операторы **new** и **delete** перегружаются для класса **loc**.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);
};

// Оператор new, перегруженный для оператора loc.
void *loc::operator new(size_t size)
{
  void *p;

  cout << "Внутри перегруженного оператора new.\n";
  p =  malloc(size);
```

```
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Оператор delete, перегруженный для класса loc.
void loc::operator delete(void *p)
{
  cout << "Внутри перегруженного оператора delete.\n";
  free(p);
}

int main()
{
  loc *p1, *p2;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Ошибка при выделении памяти для объекта p1.\n";
    return 1;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Ошибка при выделении памяти для объекта p2.\n";
    return 1;;
  }

  p1->show();
  p2->show();

  delete p1;
  delete p2;

  return 0;
}
```

Эта программа выводит на экран такие строки.

```
Внутри перегруженного оператора new.
Внутри перегруженного оператора new.
10 20
-10 -20
Внутри перегруженного оператора delete.
Внутри перегруженного оператора delete.
```

Если операторы **new** и **delete** перегружаются для конкретного класса, то их применение к другим классам будет означать вызов стандартных операторов **new** и **delete**. Перегруженные операторы применяются только к тем типам, для которых они определены. Это значит, что в следующей строке программы будет выполнен стандартный оператор **new**.

```
int *f = new float; // Выполняется стандартный оператор new
```

Если операторы **new** и **delete** перегрузить вне классов, они будут считаться глобальными. В этом случае стандартные операторы **new** и **delete** игнорируются.

Разумеется, если какие-то классы содержат свои версии этих операторов, то будут выполнены именно они. Иначе говоря, в каждом конкретном случае компилятор сам анализирует, какой вариант оператора **new** или **delete** следует выполнить. Если обнаруживается перегруженный вариант оператора, определенный для конкретного класса, то вызывается именно эта версия, в противном случае применяется глобальный оператор. Если глобальный оператор перегружен, выполняются его перегруженные версии.

Рассмотрим пример, в котором операторы **new** и **delete** перегружены глобально.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
};

// Глобальный оператор new
void *operator new(size_t size)
{
  void *p;

  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Глобальный оператор delete
void operator delete(void *p)
{
  free(p);
}

int main()
{
  loc *p1, *p2;
  float *f;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Ошибка при выделении памяти для объекта p1.\n";
```

```
    return 1;;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << " Ошибка при выделении памяти для объекта p2.\n";
    return 1;;
  }

  try {
    f = new float; // Ипользуется перегруженная версия
                   // оператора new
  } catch (bad_alloc xa) {
    cout << " Ошибка при выделении памяти для объекта f.\n";
    return 1;;
  }

  *f = 10.10F;
  cout << *f << "\n";

  p1->show();
  p2->show();

  delete p1;
  delete p2;
  delete f;

  return 0;
}
```

Запустите эту программу и убедитесь, что для встроенных типов действительно вызываются перегруженные операторы **new** и **delete**.

Перегрузка операторов new и delete для массивов

Если вы хотите определить свою собственную систему распределения памяти для массивов, необходимо еще раз перегрузить операторы **new** и **delete** так, как показано ниже.

```
// Размещение массива объектов.
void *operator new[](size_t size)
{
  /* Выделяется память.  При отказе генерируется
     исключительная ситуация bad_alloc.
     Конструктор каждого элемента вызывается автоматически. */
  return pointer_to_memory;
}

// Удаление массива объектов.
void operator delete[](void *p)
{
  /* Освобождается память, на которую ссылается указатель p.
     Деструктор каждого элемента вызывается автоматически.
  */
}
```

После размещения массива в памяти для создания каждого из его элементов автоматически вызывается конструктор. При освобождении памяти для уничтожения каждого объекта автоматически вызывается деструктор. Для этих действий не обязательно предусматривать отдельный код.

Следующая программа размещает в памяти массив объекта класса `loc`, а затем освобождает занятую память.

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {longitude = latitude = 0;}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);

  void *operator new[](size_t size);
  void operator delete[](void *p);
};

// Перегруженный оператор new для класса
void loc::operator new(size_t size)
{
void *p;

  cout << "Внутри перегруженного оператора new.\n";
  p =  malloc(size);
  if(!p)
  {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Перегруженный оператор delete для класса loc.
void loc::operator delete(void *p)
{
  cout << "Внутри перегруженного оператора delete.\n";
  free(p);
}

// Перегруженный оператор new для размещения в памяти
// массива объектов класса loc.
```

```
void *loc::operator new[](size_t size)
{
  void *p;

  cout << "Применение перегруженного оператора new[].\n";
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Перегруженный оператор new для размещения в памяти
// массива объектов класса loc.
void loc::operator delete[](void *p)
{
  cout << "Удаление массива с помощью оператора delete[]\n";
  free(p);
}

int main()
{
  loc *p1, *p2;
  int i;

  try {
    p1 = new loc (10, 20); // Размещаем объект в памяти
  } catch (bad_alloc xa) {
    cout << "Ошибка при выделении памяти для объекта p1.\n";
    return 1;;
  }

  try {
    p2 = new loc [10]; // Размещаем массив в памяти
  } catch (bad_alloc xa) {
    cout << " Ошибка при выделении памяти для объекта p2.\n";
    return 1;;
  }

  p1->show();

  for(i=0; i<10; i++)
    p2[i].show();

  delete p1; // Уничтожаем объект
  delete [] p2; // Уничтожаем массив

  return 0;
}
```

Перегрузка операторов new и delete, не генерирующих исключительной ситуации

Рассмотрим перегрузку операторов **new** и **delete**, не генерирующих исключительной ситуации.

```
// Версия оператора new, не генерирующего исключительной ситуации.
void *operator new(size_t size, const nothrow_t &n)
{
  // Выделяем память.
  if(успешно) return указатель_на_область_памяти;
  else return 0;
}

// Версия оператора new, не генерирующего исключительной ситуации,
для размещения массивов.
void *operator new[](size_t size, const nothrow_t &n)
{
  // Выделяем память.
  if(успешно) return указатель_на_область_памяти;
  else return 0;
}

void operator delete(void *p, const nothrow_t &n)
{
  // освобождаем память
}

void operator delete[](void *p, const nothrow_t &n)
{
  // Освобождаем память
}
```

Тип **nothrow_t** определен в заголовке **<new>**. Он является типом объектов **nothrow**. Параметры типа **nothrow_t** не используются.

Перегрузка некоторых специальных операторов

В языке C++ существуют операторы индексирования элементов массива "**[]**", вызова функции "**()**" и доступа к члену класса "**->**". Эти операторы также можно перегружать, что порождает массу интересных возможностей.

На перегрузку этих операторов распространяется одно общее ограничение: они должны быть нестатическими функциями-членами. Дружественные функции применять нельзя.

Перегрузка оператора "[]"

При перегрузке оператор "**[]**" считается бинарным. Следовательно, он должен иметь такой вид:

```
тип имя_класса::operator[](int i)
{
  // ...
}
```

С технической точки зрения параметр не обязан иметь тип **int**, однако функция **operator[]()** обычно применяется для индексирования элементов массива, поэтому чаще всего используют именно целочисленный параметр.

Допустим, что объект называется **o**. Тогда выражение

```
o[3]
```

преобразуется в следующий вызов функции **operator[]()**:

```
O.operator[](3)
```

Иначе говоря, значение выражения, заключенного в квадратные скобки, передается функции **operator[]()** в качестве явного параметра. Указатель **this** ссылается на объект **O**, который генерирует вызов функции.

В следующей программе класс **atype** содержит массив, состоящий из трех элементов. Его конструктор инициализирует каждый элемент массива отдельным значением. Перегруженная операторная функция **operator()[]** возвращает значение элемента массива по заданному индексу.

```
#include <iostream>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // Выводит на экран число 2

  return 0;
}
```

Функцию **operator()[]** можно определить таким образом, чтобы оператор "**[]**" был допустимым и в левой, и в правой части оператора присваивания. Для этого нужно, чтобы функция **operator[]()** возвращала ссылку. Продемонстрируем это с помощью следующей программы.

```
#include <iostream>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);
```

```
  cout << ob[1]; // Выводит на экран число 2
  cout << " ";

  ob[1] = 25; // Оператор [] стоит слева

  cout << ob[1]; // Теперь на экран выводится число 25

  return 0;
}
```

Поскольку теперь операторная функция **operator[]()** возвращает ссылку на элемент массива с индексом **i**, ее можно использовать в левой части оператора присваивания для модификации элементов массива. (Разумеется, оператор "**[]**" по-прежнему можно использовать в правой части оператора присваивания.)

Перегрузка оператора "**[]**" позволяет обеспечить контроль за выходом индекса массива за пределы допустимого диапазона, который по умолчанию не предусмотрен в языке C++. Создав класс, содержащий массив, и предусмотрев перегруженный оператор "**[]**", можно перехватить возникающую исключительную ситуацию. Рассмотрим пример, в котором программа предусматривает проверку диапазона индексов массива.

```
// Пример безопасного массива.
#include <iostream>
#include <cstdlib>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i);
};

// Проверка диапазона изменения индекса для класса atype.
int &atype::operator[](int i)
{
  if(i<0 || i> 2)
  {
    cout << "Выход за пределы допустимого диапазона\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // Выводит на экран число 2
  cout << " ";

  ob[1] = 25; // Оператор [] слева
  cout << ob[1]; // Выводит на экран число 25
```

```
  ob[3] = 44; // Генерируется ошибка, значение 3
              // лежит вне допустимого диапазона

  return 0;
}
```

При выполнении оператора

```
ob[3] = 44;
```

функция **operator[]()** перехватывает возникшую исключительную ситуацию и прекращает выполнение программы. (На практике выполнение программы обычно не прекращается, а вместо этого вызывается обработчик исключительной ситуации, связанной с выходом индекса за пределы допустимого диапазона.)

Перегрузка оператора "()"

При перегрузке оператора "**()**" создается операторная функция, которой можно передавать произвольное количество параметров. При этом новый способ вызова функции на самом деле не создается. Рассмотрим пример. Допустим, что оператор вызова функции перегружен в некотором классе следующим образом:

```
double operator()(int a, float f, char *s);
```

Кроме того, предположим, что переменная **O** является объектом этого класса. Тогда оператор

```
O(10, 23.34, "Привет");
```

преобразуется в следующий вызов операторной функции **operator()**:

```
O.operator()(10, 23.34, "Привет");
```

Как правило, при перегрузке оператора "**()**" определяются параметры, передаваемые функции. При вызове функции эти параметры копируются и присваиваются соответствующим аргументам. Как всегда, указатель **this** ссылается на объект, генерирующий вызов операторной функции (в данном примере на объект **O**).

Рассмотрим пример перегруженного оператора "**()**" в классе **loc**. Он присваивает значения двух своих аргументов членам объекта класса **loc**, в которых хранится значение широты и долготы.

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
```

```
  loc operator+(loc op2);
  loc operator()(int i, int j);
};

// Перегруженный оператор ( ) для класса loc.
loc loc::operator()(int i, int j)
{
  longitude = i;
  latitude = j;

  return *this;
}

// Перегруженный оператор + для класса loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2(1, 1);

  ob1.show();
  ob1(7, 8); // Оператор применяется самостоятельно
  ob1.show();

  ob1 = ob2 + ob1(10, 10); // Оператор можно использовать
                           // внутри выражений
  ob1.show();

  return 0;
}
```

Эта программа выводит на экран такие строки.

```
10 20
7 8
11 11
```

Помните, что, перегружая оператор "`()`", вы можете использовать любой тип параметров и любой тип возвращаемого значения. Эти типы можно выбирать в зависимости от конкретного приложения. Кроме того, можно предусматривать значения аргументов по умолчанию.

Перегрузка оператора "->"

Оператор *ссылки на член объекта* (class member access operator) при перегрузке считается унарным. Его общий вид таков.

```
объект -> элемент
```

Операторная функция, перегружающая оператор "->", вызывается из *объекта*. Функция **operator->()** должна возвращать указатель на объект класса, для которого он определен. *Элемент* должен быть членом, доступным внутри объекта.

Следующая программа иллюстрирует перегрузку оператора "->" и демонстрирует эквивалентность выражений **ob.i** и **ob->i**, когда операторная функция **operator->()** возвращает указатель **this**.

```
#include <iostream>
using namespace std;

class myclass {
public:
  int i;
  myclass *operator->() {return this;}
};

int main()
{
  myclass ob;

  ob->i = 10; // Эквивалентно выражению ob.i

  cout << ob.i << " " << ob->i;

  return 0;
}
```

Операторная функция **operator->()** должна быть членом класса, для которого она определяется.

Перегрузка оператора ","

Язык C++ допускает перегрузку оператора ",". Этот оператор является бинарным. Его смысл при перегрузке может быть произвольным. Однако, если целью перегрузки является выполнение операции, аналогичной стандартной, следует игнорировать все аргументы, кроме крайнего справа. Именно этот параметр считается результатом стандартного оператора ",".

Рассмотрим программу, демонстрирующую перегрузку оператора ",".

```
#include <iostream>
using namespace std;

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
```

```
  loc operator+(loc op2);
  loc operator,(loc op2);
};

// Перегрузка оператора "запятая" для класса loc
loc loc::operator,(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude;
  temp.latitude = op2.latitude;
  cout << op2.longitude << " " << op2.latitude << "\n";

  return temp;
}

// Перегрузка оператора + для класса loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);

  ob1.show();
  ob2.show();
  ob3.show();
  cout << "\n";

  ob1 = (ob1, ob2+ob2, ob3);

  ob1.show(); // Выводит на экран числа 1 1,
              // т.е. значение объекта ob3

  return 0;
}
```

Эта программа выводит на экран следующие строки.

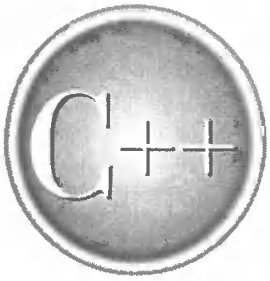
```
10 20
5 30
1 1

10 60
1 1
1 1
```

Несмотря на то что все операнды, стоящие в левой части, игнорируются, каждое выражение по-прежнему вычисляется компилятором, поэтому возможны все предусмотренные побочные эффекты.

Следует помнить, что операнд, стоящий в левой части, передается с помощью указателя `this`, а его значение игнорируется функцией `operator,()`. Функция возвращает значение операнда, стоящего в правой части. Таким образом, действия перегруженного оператора "`,`" напоминают стандартные. Если вы хотите, чтобы оператор выполнял другие операции, следует изменить эти свойства.

Полный справочник по



Глава 16

Наследование

Наследование — один из краеугольных камней объектно-ориентированного программирования, так как оно позволяет создавать иерархические классификации. Используя наследование, можно создавать общие классы, определяющие свойства, характерные для всей совокупности родственных классов. Эти классы могут наследовать свойства друг у друга, добавляя к ним свои собственные уникальные характеристики.

Согласно стандартной терминологии языка C++ класс, лежащий в основе иерархии, называется *базовым* (base class), а класс, наследующий свойства базового класса, — *производным* (derived class). Производные классы, в свою очередь, могут быть базовыми по отношению к другим классам.

В языке C++ предусмотрен мощный и гибкий механизм наследования. Первичные сведения о наследовании содержатся в главе 11. Настало время изучить его подробнее.

Управление доступом к членам базового класса

При наследовании члены базового класса становятся членами производного класса. Как правило, для наследования используется следующая синтаксическая конструкция.

```
class имя-производного-класса:уровень_доступа имя-базового-класса{
  // тело класса
  };
```

Параметр *уровень_доступа* определяет статус членов базового класса в производном классе. В качестве этого параметра используются спецификаторы **public**, **private** или **protected**. Если уровень доступа не указан, то для производного класса по умолчанию используется спецификатор **private**, а для производной структуры — **public**. Рассмотрим варианты, возникающие в этих ситуациях. (Спецификатор **protected** будет описан в следующем разделе.)

Если уровень доступа к членам базового класса задается спецификатором **public**, то все открытые и защищенные члены базового класса становятся открытыми и защищенными членами производного класса. При этом закрытые члены базового класса не меняют своего статуса и остаются недоступными членам производного. Как демонстрирует следующая программа, объекты класса **derived** могут непосредственно ссылаться на открытые члены класса **base**.

```
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};
```

```
int main()
{
  derived ob(3);

  ob.set(1, 2); // Обращение к члену класса base
  ob.show(); // Обращение к члену класса base

  ob.showk(); // Обращение к члену класса derived

  return 0;
}
```

Если свойства базового класса наследуются с помощью спецификатора доступа **private**, все открытые и защищенные члены базового класса становятся закрытыми членами производного класса. Например, следующая программа даже не будет скомпилирована, так как обе функции **set()** и **show()** теперь являются закрытыми членами класса **derived**.

```
// Эта программа не будет скомпилирована.
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n";}
};

// Открытые члены класса base являются
// закрытыми членами класса derived.
class derived : private base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // Ошибка, доступ к функции set() запрещен.
  ob.show(); // Ошибка, доступ к функции show() запрещен.

  return 0;
}
```

Внимание! *При закрытом наследовании все открытые и защищенные члены базового класса становятся закрытыми членами производного класса. Это значит, что они остаются доступными членам производного класса, но недоступны остальным элементам программы, не являющимся членами базового или производного классов.*

Наследование и защищенные члены

Спецификатор **protected** повышает гибкость механизма наследования. Если член класса объявлен защищенным (protected), то вне класса он недоступен. С этой точки зрения защищенный член класса ничем не отличается от закрытого. Единственное

исключение из этого правила касается наследования. В этой ситуации защищенный член класса существенно отличается от закрытого.

Как указывалось в предыдущем разделе, закрытый член базового класса не доступен другим элементам программы, включая производный класс. Однако защищенные члены базового класса ведут себя иначе. При открытом наследовании защищенные члены базового класса становятся защищенными членами производного класса и, следовательно, доступны остальным членам производного класса. Иными словами, защищенные члены класса по отношению к своему классу являются закрытыми и, в то же время, могут наследоваться производным классом. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j; // Закрыты по отношению к классу base,
            // но доступны классу derived.
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  // Класс derived имеет доступ к членам i и j из класса base
  void setk() { k=i*j; }

  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob;

  ob.set(2, 3); // Все в порядке, этот член доступен классу derived
  ob.show(); // Все в порядке, этот член доступен классу derived

  ob.setk();
  ob.showk();

  return 0;
}
```

В данном примере, поскольку класс **derived** наследует свойства класса **base** с помощью открытого наследования, а переменные **i** и **j** объявлены защищенными, функция **seek()** из класса **derived** имеет к ним доступ. Если бы переменные **i** и **j** были объявлены в классе **base** закрытыми, то класс **derived** не имел бы к ним доступа, и программу нельзя было скомпилировать.

Если производный класс является базовым по отношению к другому производному классу, то любой защищенный член исходного базового класса, открыто наследуемый первым производным классом, также может наследоваться вторым производным классом как защищенный член. Например, следующая программа вполне корректна, и класс **derived2** действительно имеет доступ к переменным **i** и **j**.

```
#include <iostream>
using namespace std;
```

```
class base {
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// Переменные i и j наследуются как защищенные.
class derived1 : public base {
  int k;
public:
  void setk() { k = i*j; } // legal
  void showk() { cout << k << "\n"; }
};

// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
  int m;
public:
  void setm() { m = i-j; } // Допускается
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(2, 3);
  ob1.show();
  ob1.setk();
  ob1.showk();

  ob2.set(3, 4);
  ob2.show();
  ob2.setk();
  ob2.setm();
  ob2.showk();
  ob2.showm();

  return 0;
}
```

Однако, если бы к классу **base** применялся механизм закрытого наследования, то все его члены стали бы закрытыми членами класса **derived1** и были недоступны классу **derived2**. (В то же время переменные **i** и **j** были бы по-прежнему доступны классу **derived1**.) Эта ситуация иллюстрируется следующей программой (она содержит ошибку и не компилируется).

```
// Эта программа содержит ошибку.
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
```

```
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// Теперь все члены класса base являются закрытыми членами
// класса derived1.
class derived1 : private base {
  int k;
public:
  // Это делать можно, поскольку переменные i и j
  // являются закрытыми членами класса derived1.
  void setk() { k = i*j; } // OK
  void showk() { cout << k << "\n"; }
};

// Доступ к членам i, j, set() и show() не наследуется.
class derived2 : public derived1 {
  int m;
public:
  // Неправильно, так как переменные i и j являются закрытыми
  // членами класса derived1
  void setm() { m = i-j; } // Ошибка
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // Ошибка, вызывать функцию set() нельзя.
  ob1.show(); // Ошибка, вызывать функцию show() нельзя.

  ob2.set(3, 4); // Ошибка, вызывать функцию set() нельзя.
  ob2.show(); // Ошибка, вызывать функцию show() нельзя.

  return 0;
}
```

На заметку *Даже если бы к классу **base** применялось закрытое наследование, класс **derived1** по-прежнему имел бы доступ к его открытым и защищенным членам. Однако эти привилегии другим наследникам не передаются.*

Защищенное наследование

К базовому классу можно применять механизм защищенного наследования. При этом все открытые и защищенные члены базового класса становятся защищенными членами производного класса. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class base {
protected:
  int i, j; // Закрытые члены класса base,
            // доступные классу derived.
```

```
public:
  void setij(int a, int b) { i=a; j=b; }
  void showij() { cout << i << " " << j << "\n"; }
};

// Класс, полученный с помощью защищенного наследования.
class derived : protected base {
  int k;
public:
  // Класс derived имеет доступ к членам i, j и setij()
  // из класса base.
  void setk() { setij(10, 12); k = i*j; }

  // Отсюда можно вызвать функцию showij().
  void showall() { cout << k << " "; showij(); }
};

int main()
{
  derived ob;

//  ob.setij(2, 3); // Неверно, функция setij() является
//                     закрытым членом класса derived.

  ob.setk(); // Верно, вызывается открытый член класса derived.
  ob.showall(); // Верно, вызывается открытый член класса derived.

// ob.showij(); // Неверно, функция showij() является
                // защищенным членом класса derived

  return 0;
}
```

Как следует из комментариев, несмотря на то что функции **setij()** и **showij()** являются открытыми членами класса **base**, в классе **derived**, образованном с помощью защищенного наследования, они становятся защищенными. Это значит, что в функции **main()** они не доступны.

Множественное наследование

Производный класс может одновременно наследовать свойства нескольких базовых классов. Например, в программе, приведенной ниже, класс **derived** наследует свойства классов **base1** и **base2**.

```
// Пример множественного наследования.

#include <iostream>
using namespace std;

class base1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};
```

```
class base2 {
protected:
  int y;
public:
  void showy() {cout << y << "\n";}
};

// Множественное наследование.
class derived: public base1, public base2 {
public:
  void set(int i, int j) { x=i; y=j; }
};

int main()
{
  derived ob;

  ob.set(10, 20); // Эта функция принадлежит классу derived.
  ob.showx(); // Эта функция принадлежит классу base1.
  ob.showy(); // Эта функция принадлежит классу base2.

  return 0;
}
```

Как видим, при множественном наследовании имена базовых классов перечисляются в списке и разделяются запятыми, причем перед каждым именем базового класса указывается свой спецификатор доступа.

Конструкторы, деструкторы и наследование

В связи с наследованием возникают два вопроса, касающиеся конструкторов и деструкторов. Во-первых, когда вызываются конструкторы и деструкторы базового и производного классов? Во-вторых, как передаются параметры конструкторов базового класса? Ответы на эти вопросы содержатся в следующем разделе.

Когда вызываются конструкторы и деструкторы

Базовый и производный класс могут содержать несколько конструкторов и деструктор. Следовательно, очень важно правильно понимать, в каком порядке они вызываются при создании и уничтожении объектов производного класса. Для начала рассмотрим следующий пример.

```
#include <iostream>
using namespace std;

class base {
public:
  base() { cout << "Создается объект класса base\n"; }
  ~base() { cout << "Уничтожается объект класса base\n"; }
};

class derived: public base {
public:
  derived() { cout << "Создается объект класса derived\n"; }
  ~derived() { cout << "Уничтожается объект класса derived\n"; }
};
```

```
int main()
{
  derived ob;

  // Кроме создания и уничтожения объекта, ничего не происходит

  return 0;
}
```

Как указано в комментарии к функции **main()**, программа просто создает, а затем уничтожает объект **ob** класса **derived**. В ходе выполнения программа выводит на экран следующие сообщения.

```
Создание объекта класса base
Создание объекта класса derived
Уничтожение объекта класса derived
Уничтожение объекта класса base
```

Как видим, сначала вызывается конструктор базового класса, а затем — производного. После этого, поскольку объект **ob** немедленно уничтожается, вызывается деструктор класса **derived**, а за ним — деструктор класса **base**.

Результаты этого эксперимента можно обобщить. При создании объекта производного класса сначала вызывается конструктор базового класса, а потом — производного. При уничтожении объекта производного класса сначала вызывается деструктор производного класса, а затем — базового. Иначе говоря, конструкторы вызываются в иерархическом порядке, а деструкторы — в обратном.

Это вполне естественно. Поскольку базовый класс не имеет никакой информации о производных классах, инициализация его объектов должна выполняться до инициализации любого объекта производного класса. Следовательно, конструктор базового класса должен вызываться первым.

Вполне очевидно, что деструкторы должны вызываться в обратном порядке. Поскольку производный класс наследует свойства базового, уничтожение объекта базового класса вызовет уничтожение объекта производного класса. Следовательно, деструктор производного класса должен вызываться до полного уничтожения объекта.

При иерархическом наследовании (когда производный класс становится базовым для своего наследника) применяется следующее правило: конструкторы вызываются в иерархическом порядке, а деструкторы — в обратном. Рассмотрим пример.

```
#include <iostream>
using namespace std;

class base {
public:
  base() { cout << "Создание объекта класса base\n"; }
  ~base() { cout << "Уничтожение объекта класса base\n"; }
};

class derived1 : public base {
public:
  derived1() { cout << "Создание объекта класса derived1\n"; }
  ~derived1() { cout << "Уничтожение объекта класса derived1\n"; }
};

class derived2: public derived1 {
public:
  derived2() { cout << "Создание объекта класса derived2\n"; }
  ~derived2() { cout << "Уничтожение объекта класса derived2\n"; }
};
```

```
int main()
{
  derived2 ob;

  // Создаем и уничтожаем объект ob

  return 0;
}
```

В результате на экран выводятся следующие строки.

```
Создание объекта класса base
Создание объекта класса derived1
Создание объекта класса derived2
Уничтожение объекта класса derived2
Уничтожение объекта класса derived1
Уничтожение объекта класса base
```

Это правило применимо и к множественному наследованию. Рассмотрим следующую программу.

```
#include <iostream>
using namespace std;

class base1 {
public:
  base1() { cout << "Создание объекта класса base1\n"; }
  ~base1() { cout << "Уничтожение объекта класса base1\n"; }
};

class base2 {
public:
  base2() { cout << " Создание объекта класса base2\n"; }
  ~base2() { cout << " Создание объекта класса base2\n"; }
};

class derived: public base1, public base2 {
public:
  derived() { cout << " Создание объекта класса derived\n"; }
  ~derived() { cout << " Уничтожение объекта класса derived\n"; }
};

int main()
{
  derived ob;

  // Создание и уничтожение объекта ob

  return 0;
}
```

Эта программа выдает на экран следующие сообщения.

```
Создание объекта класса base1
Создание объекта класса base2
Создание объекта класса derived
Уничтожение объекта класса derived
Уничтожение объекта класса base2
Уничтожение объекта класса base1
```

Как видим, и в этом случае конструкторы вызываются в иерархическом порядке, слева направо, как указано в списке наследования класса **derived**. Деструкторы вызываются в обратном порядке, справа налево. Допустим, что имя **base2** указано в списке наследования класса **derived** перед именем **base1**.

```
class derived: public base2, public base1 {
```

Тогда результаты работы программы выглядели бы так.

```
Создание объекта класса base2
Создание объекта класса base1
Создание объекта класса derived
Уничтожение объекта класса derived
Уничтожение объекта класса base1
Уничтожение объекта класса base2
```

Передача параметров конструктору базового класса

До сих пор мы рассматривали конструкторы, не имеющие аргументов. Если конструктор производного класса должен получать несколько параметров, следует просто использовать стандартную синтаксическую форму конструктора с параметрами (см. главу 12). Однако возникает вопрос, каким образом передаются аргументы конструктору базового класса? Для этого применяется расширенная форма объявления конструктора производного класса, которая позволяет передавать аргументы нескольким конструкторам одного или нескольких базовых классов. Общая форма этой синтаксической конструкции такова.

```
конструктор_производного-класса (список_аргументов) :base1(список_аргументов) ,
                                                      base2(список_аргументов) ,
                                                      ...
                                                      baseN(список_аргументов)
{
    // Тело конструктора производного класса
}
```

Здесь параметры *base1*–*baseN* являются именами базовых классов. Обратите внимание на то, что объявление конструктора производного класса отделяется двоеточием от спецификаций базовых классов, которые, в свою очередь, разделяются запятыми. Рассмотрим следующую программу.

```
#include <iostream>
using namespace std;

class base {
protected:
  int i;
public:
  base(int x) { i=x; cout << "Создание объекта класса base\n"; }
  ~base() { cout << "Уничтожение объекта класса base\n"; }
};

class derived: public base {
  int j;
public:
  // Класс derived использует переменную x;
  // переменная y передается базовому классу.
  derived(int x, int y): base(y)
    { j=x; cout << "Создание объекта класса derived\n"; }
```

```
  ~derived() { cout << "Уничтожение объекта класса derived\n"; }
  void show() { cout << i << " " << j << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // Выводит на экран числа 4 3

  return 0;
}
```

Здесь конструктор класса **derived** имеет два параметра: **x** и **y**. Однако в самом конструкторе используется лишь переменная **x**, а переменная **y** передается конструктору базового класса. Как правило, в конструкторе производного класса должны объявляться все параметры, необходимые базовому классу Для этого они указываются после двоеточия в списке аргументов конструктора базового класса.

Рассмотрим пример множественного наследования.

```
#include <iostream>
using namespace std;

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Создание объекта класса base1\n"; }
  ~base1() { cout << "Уничтожение объекта класса base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Создание объекта класса base2\n"; }
  ~base2() { cout << "Уничтожение объекта класса base2\n"; }
};

class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Создание объекта класса derived\n"; }

  ~derived() { cout << "Уничтожение объекта класса derived\n"; }
  void show() { cout << i << " " << j << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4, 5);

  ob.show(); // Выводит на экран числа 4 3 5

  return 0;
}
```

Подчеркнем, что аргументы конструктора базового класса передаются с помощью аргументов конструктора производного класса. Следовательно, даже если конструктор производного класса не имеет собственных аргументов, его объявление должно содержать аргументы конструкторов базовых классов. В этом случае аргументы, передаваемые конструктору производного класса, просто переправляются конструкторам базовых классов. Например, в рассмотренной ниже программе конструктор класса **deroved** не имеет собственных аргументов, а конструкторы класса **base1** и **base2**, напротив, имеют по одному параметру.

```
#include <iostream>
using namespace std;

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Создание объекта класса base1\n"; }
  ~base1() { cout << "Уничтожение объекта класса base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Создание объекта класса base2\n"; }
  ~base2() { cout << "Уничтожение объекта класса base2\n"; }
};

class derived: public base1, public base2 {
public:
  /* Конструктор класса derived не имеет параметров,
 в его объявлении указываются параметры конструкторов
     базовых классов.
 */

  derived(int x, int y): base1(x), base2(y)
    { cout << "Создание объекта класса derived\n"; }

  ~derived() { cout << "Уничтожение объекта класса derived\n"; }
  void show() { cout << i << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // Выводит на экран числа 3 4

  return 0;
}
```

Конструктор производного класса может произвольно использовать все параметры, указанные в его объявлении, даже если они передаются конструкторам базового класса. Иначе говоря, передача параметров конструкторам базовых классов не исключает их использования внутри производного класса. Таким образом, фрагмент программы, приведенный ниже, является абсолютно правильным.

```
class derived: public base {
  int j;
```

```
public:
  // Класс derived использует оба параметра x и y,
  // а затем передает их конструктору базового класса.
  derived(int x, int y): base(x, y)
    { j = x*y; cout << "Создание объекта класса derived\n"; }
```

Передавая параметры конструкторам базовых классов, следует иметь в виду, что в качестве аргумента могут использоваться любые допустимые выражения, например, вызовы функций или переменные. Это полностью согласуется с принципом динамической инициализации объектов, предусмотренной в языке C++.

Предоставление доступа

Если к базовому классу применяется механизм закрытого наследования, все его открытые и защищенные члены становятся закрытыми членами производного класса. Однако в некоторых ситуациях можно восстановить исходный статус одного или нескольких унаследованных членов, которые ранее были открытыми или защищенными. Например, может возникнуть необходимость сохранить открытый доступ к некоторым членам базового класса, несмотря на то что они наследуются закрытым производным классом. Стандарт языка C++ предусматривает для этого два пути. Во-первых, можно применить оператор **using**. Этот способ более предпочтителен. Оператор **using** предназначен для поддержки пространств имен и обсуждается в главе 23. Во-вторых, можно использовать *объявление уровня доступа* (access declaration) в производном классе. Этот способ также поддерживается стандартом языка C++, однако считается нежелательным. Это значит, что в новых программах его следует избегать. Однако, поскольку в обиходе остается большое количество старых программ, рассмотрим этот пример подробнее.

Объявление уровня доступа выглядит следующим образом.

```
базовый_класс::член;
```

Это объявление размещается внутри производного класса после заголовка соответствующего раздела. Обратите внимание на то, что тип переменной в объявлении уровня доступа указывать не следует.

```
class base {
public:
  int j; // Открытый член класса base
};

// Закрытый наследник класса base.
class derived: private base {
public:

  // Место для объявления уровня доступа.
  base::j; // Теперь переменная j снова открыта.
  .
  .
  .
};
```

Поскольку класс **derived** является закрытым наследником класса **base**, открытая переменная-член **j** из класса **base** становится закрытой переменной-членом класса **derived**. Однако в раздел **public** класса **derived** мы поместили объявление уровня доступа

```
base::j;
```

Переменная **j** снова стала открытой.

Этот способ можно применять для восстановления статуса открытых и защищенных членов. Однако с его помощью нельзя поднять или понизить уровень доступа к члену класса. Например, член, объявленный закрытым в базовом классе, нельзя сделать открытым в производном. (Если бы это было возможно, механизм инкапсуляции был бы полностью разрушен!)

Следующая программа иллюстрирует объявление уровня доступа. Обратите внимание на то, как восстанавливается открытый статус членов **j**, **seti()** и **geti()**.

```
#include <iostream>
using namespace std;

class base {
  int i; // Закрытый член класса base
public:
  int j, k;
  void seti(int x) { i = x; }
  int geti() { return i; }
};

// Закрытый наследник класса base.
class derived: private base {
public:
  /* Следующие три оператора восстанавливают
     открытый статус членов j, seti() и geti(). */
  base::j; // Переменная j снова открыта, а переменная k — нет.
  base::seti; // Функция seti() снова открыта.
  base::geti; // Функция geti() снова открыта.

// base::i; // Неверно, уровень доступа поднимать нельзя.

  int a; // Открытый член.
};

int main()
{
  derived ob;

//ob.i = 10; // Нельзя, так как переменная i
             // является закрытым членом класса derived.

  ob.j = 20; // Можно, так как переменная j открыта в классе derived.
//ob.k = 30; // Нельзя, поскольку переменная k является
             // закрытым членом класса derived.

  ob.a = 40; // Можно, так как переменная a является открытым
             // членом класса derived.
  ob.seti(10);

  cout << ob.geti() << " " << ob.j << " " << ob.a;

  return 0;
}
```

Этот способ позволяет восстанавливать уровень доступа к некоторым открытым или защищенным членам класса, оставляя производный класс закрытым.

Внимание! *Несмотря на то что язык C++ допускает объявления уровня доступа, они считаются нежелательными. Это означает, что при создании новых программ вместо них следует применять оператор* **`using`**, *обеспечивающий тот же результат.*

Виртуальные базовые классы

При множественном наследовании может возникнуть неоднозначность. Рассмотрим, например, следующую неправильную программу.

```
// Эта программа содержит ошибку и не компилируется.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// Класс derived1 является наследником класса base.
class derived1 : public base {
public:
  int j;
};

// Класс derived2 является наследником класса base.
class derived2 : public base {
public:
  int k;
};

/* Класс derived3 является наследником классов derived1
   и derived2. Следовательно, в классе derived3 существуют
   две копии класса base! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // Неоднозначность, какая переменная i имеется в виду???
  ob.j = 20;
  ob.k = 30;

  // Здесь переменная i также определена неоднозначно.
  ob.sum = ob.i + ob.j + ob.k;

  // Здесь переменная i также определена неоднозначно.
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Как указано в комментариях, классы **derived1** и **derived2** являются наследниками класса **base**. Однако класс **deribed3** является производным от обоих классов **derived2** и **derived3**. (Такое наследование называется *бриллиантовым. — Прим. ред.*) Следовательно, в объекте класса **derived3** содержатся две копии объекта класса **base**. Таким образом, выражение

```
ob.i = 20;,
```

в котором происходит обращение к переменной **i**, становится неоднозначным, поскольку неизвестно, из объекта какого класса следует взять эту переменную: **derived1** или **derived2**. Обладая двумя копиями объекта класса **base**, объект класса **ob** содержит два экземпляра переменной **ob.i**! Как видим, этот оператор в принципе неоднозначен.

Эту программу можно исправить двумя способами. Во-первых, к переменной **i** можно применить оператор разрешения области видимости. Например, следующая программа работает совершенно правильно.

```
// В этой программе для явного выбора переменной i
// применяется оператор разрешения области видимости.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// Класс derived1 является наследником класса base.
class derived1 : public base {
public:
  int j;
};

// Класс derived2 является наследником класса base.
class derived2 : public base {
public:
  int k;
};

/* Класс derived3 является наследником классов derived1
   и derived2 одновременно. Следовательно, в каждом его объекте
   содержатся две копии объекта класса base! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.derived1::i = 10; // Неоднозначность устранена,
                       // используется переменная i из класса de-
rived1.
  ob.j = 20;
  ob.k = 30;

  // Неоднозначность устранена
```

```
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // Неоднозначность устранена
  cout << ob.derived1::i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Как видим, оператор разрешения области видимости "**::**" позволяет явно выбрать вариант производного класса. Однако это решение порождает новые проблемы. Что если на самом деле нужна лишь одна копия объекта класса **base**? Можно ли предотвратить дублирование объектов класса **base** в объекте класса **derived3**? На оба вопроса можно ответить положительно. Решение этих проблем основано на применении *виртуальных базовых классов* (virtual base classes).

Если базовый класс имеет несколько наследников, его дублирование можно предотвратить. Для этого в объявлении производного класса перед именем базового класса следует поставить ключевое слово **virtual**. Например, предыдущую программу можно исправить следующим образом.

```
// Программа, использующая виртуальный базовый класс.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// Класс derived1 является наследником виртуального класса base.
class derived1 : virtual public base {
public:
  int j;
};

// Класс derived1 является наследником виртуального класса base.
class derived2 : virtual public base {
public:
  int k;
};

/* Класс derived3 является наследником классов derived1
   и derived2. На этот раз его объект содержит лишь одну
   копию объекта базового класса. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // Неоднозначность устранена
  ob.j = 20;
```

```
  ob.k = 30;

  // Неоднозначность устранена
  ob.sum = ob.i + ob.j + ob.k;

  // Неоднозначность устранена
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Как видим, перед именем базового класса в спецификации производного класса стоит ключевое слово **virtual**. Теперь оба класса **derived1** и **derived2** являются наследниками виртуального базового класса **base**, и любые их наследники будут содержать лишь одну копию класса **base**. Следовательно, объект класса **derived3** содержит копию объекта класса **base**, и выражение **ob.i=10** становится совершенно правильным и однозначным.

Следует иметь в виду, что, хотя классы **derived1** и **derived2** объявили класс **base** виртуальным, его объекты будут по-прежнему частью объектов любого из этих типов. Например, следующий фрагмент является совершенно правильным.

```
// Определяем объект класса derived1
derived1 myclass;

myclass.i = 88;
```

Различие между обычным базовым классом и виртуальным проявляется, только когда объект наследует несколько объектов одного и того же базового класса. Если используется виртуальный базовый класс, то его объект только один раз копируется в каждый объект производного класса. В противном случае возникает неоднозначность.

Полный справочник по

Глава 17

Виртуальные функции и полиморфизм

Язык C++ обеспечивает как статический, так и динамический полиморфизм. Как указывалось в предыдущих главах, статический полиморфизм достигается с помощью перегрузки функций и операторов. Динамический полиморфизм реализуется на основе наследования и виртуальных функций. Именно эта тема находится в центре внимания данной главы.

Виртуальные функции

Виртуальная функция (virtual function) — это функция-член, объявленная в базовом классе и переопределенная в производном. Чтобы создать виртуальную функцию, следует указать ключевое слово **virtual** перед ее объявлением в базовом классе. Производный класс переопределяет эту функцию, приспосабливая ее для своих нужд. По существу, виртуальная функция реализует принцип "один интерфейс, несколько методов", лежащий в основе полиморфизма. Виртуальная функция в базовом классе определяет *вид интерфейса*, т.е. способ вызова этой функции. Каждое переопределение виртуальной функции в производном классе реализует операции, присущие лишь данному классу. Иначе говоря, переопределение виртуальной функции создает *конкретный метод* (specific method).

При обычном вызове виртуальные функции ничем не отличаются от остальных функций-членов. Особые свойства виртуальных функций проявляются при их вызове с помощью указателей. Как сказано в главе 13, указатели на объекты базового класса можно использовать для ссылки на объекты производных классов. Если указатель на объект базового класса устанавливается на объект производного класса, содержащий виртуальную функцию, выбор требуемой функции основывается на *типе объекта, на который ссылается указатель*, причем этот выбор осуществляется *в ходе выполнения программы*. Таким образом, если указатель ссылается на объекты разных типов, то будут вызваны разные виртуальные функции. Это относится и к ссылкам на объекты базового класса.

Рассмотрим для начала следующий пример.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Функция vfunc() из класса base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "Функция vfunc() из класса derived1.\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
    cout << " Функция vfunc() из класса derived2.\n";
  }
};
```

```
int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // Указатель на объект базового класса.
  p = &b;
  p->vfunc(); // Вызов функции vfunc() из класса base.

  // Указатель на объект класса derived1.
  p = &d1;
  p->vfunc(); // Вызов функции vfunc() из класса derived1.

  // Указатель на объект класса derived2.
  p = &d2;
  p->vfunc(); // Вызов функции vfunc() из класса derived2.

  return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Функция vfunc() из класса base.
Функция vfunc() из класса derived1.
Функция vfunc() из класса derived2.
```

Как показывает эта программа, внутри класса **base** объявлена виртуальная функция **vfunc()**. Обратите внимание на ключевое слово **virtual** в объявлении функции. При переопределении функции **vfunc()** в классах **derived1** и **derived2** ключевое слово **virtual** не требуется. (Однако его использование не является ошибкой, просто оно не обязательно.)

В данной программе классы **derived1** и **derived2** являются производными от класса **base**. Внутри каждого из этих классов функция **vfunc()** переопределяется заново в соответствии с новым предназначением. В программе **main()** объявлены четыре переменные.

Имя	*Тип*
p	Указатель на базовый класс
b	Объект базового класса
d1	Объект класса **derived1**
d2	Объект класса **derived2**

Кроме того, указателю **p** присваивается адрес объекта **b**, а функция **vfunc()** вызывается с помощью указателя **p**. Поскольку указатель **p** ссылается на объект класса **base**, выполняется вариант функции **vfunc()** из базового класса. Затем указателю **p** присваивается адрес объекта **d1**, и функция **vfunc()** снова вызывается с его помощью. На этот раз указатель **p** ссылается на объект класса **derived1**. Следовательно, вызывается функция **derived1::vfunc()**. В результате указателю **p** присваивается адрес объекта **d2**, поэтому выражение **p->vfunc()** приводит к вызову функции **vfunc()** из класса **derived2**. Принципиально важно, что вариант вызываемой функции определяется типом объекта, на который ссылается указатель **p**. Кроме того, выбор происходит в ходе выполнения программы, что обеспечивает основу динамического полиморфизма.

Виртуальную функцию можно вызывать обычным способом, используя имя объекта и оператор "**.**", однако полиморфизм достигается только при обращении к ней через указатель. Например, следующий фрагмент программы является совершенно правильным.

```
d2.vfunc(); // Вызывается функция vfunc() из класса derived2.
```

Несмотря на то что такой вызов виртуальной функции ошибкой не является, никаких преимуществ он не предоставляет.

На первый взгляд, переопределение виртуальной функции в производном классе мало отличается от обычной перегрузки функций. Однако это не так, и термин *перегрузка* неприменим к переопределению виртуальных функций по нескольким причинам. Наиболее важное отличие заключается в том, что прототип переопределяемой виртуальной функции должен точно совпадать с прототипом, определенным в базовом классе. Этим виртуальные функции отличаются от перегруженных, которые отличаются типами и количеством параметров. (Фактически при перегрузке функций типы и количество их параметров *должны* отличаться! Именно эти отличия позволяют компилятору выбирать правильный вариант перегруженной функции.) При переопределении виртуальной функции все аспекты их прототипов должны быть одинаковыми. Если не соблюдать это правило, компилятор будет считать эти функции просто перегруженными, а их виртуальная природа будет потеряна. Второе важное ограничение заключается в том, что виртуальные функции не могут быть статическими членами классов. Кроме того, они не могут быть дружественными функциями. И, наконец, конструкторы не могут быть виртуальными, хотя на деструкторы это ограничение не распространяется.

Из-за перечисленных ограничений для переопределения виртуальной функции в производном классе используется термин *замещение* (overriding).

Вызов виртуальной функции с помощью ссылки на объект базового класса

В предыдущем примере виртуальная функция вызывалась с помощью указателя на объект базового класса, однако полиморфная природа виртуальных функций сохраняется и при их вызове с помощью ссылки на объект базового класса. Как говорилось в главе 13, ссылка является неявным указателем. Таким образом, ссылку на объект базового класса можно использовать для обращения к объекту базового или любого производного класса. Если виртуальная функция вызывается с помощью ссылки на объект базового класса, ее вариант зависит от типа объекта, на который установлена ссылка в момент вызова.

В большинстве случаев виртуальная функция с помощью ссылки вызывается при передаче параметров. Рассмотрим еще один вариант предыдущей программы.

```
/* В этой программе для вызова виртуальной функции
   применяется ссылка на объект базового класса. */
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Функция vfunc() из класса base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << " Функция vfunc() из класса derived1.\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
```

```
cout << " Функция vfunc() из класса derived2.\n";
  }
};

// Используется параметр, являющийся ссылкой на объект
// базового класса.
void f(base &r)
{
  r.vfunc();
}

int main()
{
  base b;
  derived1 d1;
  derived2 d2;

  f(b);  // Функции f() передается объект класса base.
  f(d1); // Функции f() передается объект класса derived1.
  f(d2); // Функции f() передается объект класса derived2.

  return 0;
}
```

Эта программа выводит на экран те же сообщения, что и ранее. В данном примере функция **f()** получает в качестве параметра ссылку на объект класса **base**. В функции **main()** эта функция вызывается с помощью объектов классов **base**, **derived1** и **derived2**. Конкретный вариант функции **vfunc()** выбирается внутри функции **f()** в зависимости от типа ее параметра.

Для простоты в остальных примерах этой главы виртуальные функции вызываются с помощью указателей, причем результат ничем не отличается от вызова с помощью ссылок.

Атрибут virtual наследуется

При наследовании виртуальной функции ее виртуальная природа также наследуется. Это значит, что если производный класс, унаследовавший виртуальную функцию от базового класса, становится базовым по отношению к другому производному классу, виртуальная функция может по-прежнему замещаться. Иначе говоря, не имеет значения, сколько раз наследовалась виртуальная функция, она все равно остается виртуальной. Рассмотрим следующую программу.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Функция vfunc() из класса base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << " Функция vfunc() из класса derived1.\n";
  }
};
```

```
/* Класс derived2 наследует виртуальную функцию vfunc()
   от класса derived1. */
class derived2 : public derived1 {
public:
  // Функция vfunc() остается виртуальной.
  void vfunc() {
    cout << " Функция vfunc() из класса derived2.\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // Указатель на объект класса base.
  p = &b;
  p->vfunc(); // Вызов функции vfunc() из класса base.

  // Указатель на объект класса derived1
  p = &d1;
  p->vfunc(); // Вызов функции vfunc() из класса derived1.

  // Указатель на класс derived2
  p = &d2;
  p->vfunc(); // Вызов функции vfunc() из класса derived2.

  return 0;
}
```

Как и ожидалось, программа выводит на экран следующие сообщения.

```
Вызов функции vfunc() из класса base.
Вызов функции vfunc() из класса derived1.
Вызов функции vfunc() из класса derived2.
```

В данном случае класс **derived2** является наследником класса **derived1**, а не класса **base**, но функция **vfunc()** остается виртуальной.

Виртуальные функции являются иерархическими

Как известно, если функция объявлена виртуальной в базовом классе, ее можно заместить в производном классе. Однако виртуальную функцию не обязательно замещать. В этом случае вызывается функция, определенная в базовом классе. Рассмотрим в качестве примера программу, в которой класс **derived2** не замещает функцию **vfunc()**.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "Функция vfunc() из класса base.\n";
  }
};
```

```
class derived1 : public base {
public:
  void vfunc() {
    cout << " Функция vfunc() из класса derived1.\n";
  }
};

class derived2 : public base {
public:
// Функция vfunc() не замещается в классе derived2,
// используется версия из класса base.
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // Указатель на объект класса base.
  p = &b;
  p->vfunc(); // Вызов функции vfunc() из класса base.

  // Указатель на объект класса derived1.
  p = &d1;
  p->vfunc(); // Вызов функции vfunc() из класса derived1.

  // Указатель на объект класса derived2.
  p = &d2;
  p->vfunc(); // Вызов функции vfunc() из класса base.

  return 0;
}
```

Эта программа выводит на экран следующие сообщения.

```
Функция vfunc() из класса base.
Функция vfunc() из класса derived1.
Функция vfunc() из класса base.
```

Поскольку функция **vfunc()** не замещается в классе **derived2**, через указатель на объект класса **derived2** вызывается ее версия из класса **base**.

Предыдущая программа иллюстрирует частный случай более универсального правила. Наследование в языке C++ организовано по иерархическому принципу, поэтому виртуальные функции также должны быть иерархическими. Это значит, что если виртуальная функция не замещается, вызывается ее предыдущая переопределенная версия. Например, в следующей программе класс **derived2** является наследником класса **derived1**, который, в свою очередь, является производным от класса **base**. Однако функция **vfunc()** в классе **derived2** не замещается. Следовательно, ближайшая к классу **derived2** версия функции **vfunc()** определена в классе **derived1**. Таким образом, вызов функции **vfunc()** с помощью объекта класса **derived2** относится к функции **derived1::vfunc()**.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
```

```
    cout << "Функция vfunc() из класса base.\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << " Функция vfunc() из класса derived1.\n";
  }
};

class derived2 : public derived1 {
public:
/* Функция vfunc() не замещается в классе derived2.
   Поскольку класс derived2 является наследником класса
   derived1, вызывается функция vfunc() из класса derived1.
*/
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // Указатель на объект класса base.
  p = &b;
  p->vfunc(); // Вызов функции vfunc() из класса base.

  // Указатель на объект класса derived1.
  p = &d1;
  p->vfunc(); // Функция vfunc() из класса derived1.

  // Указатель на объект класса derived2
  p = &d2;
  p->vfunc(); // Функция vfunc() из класса derived1.

  return 0;
}
```

Программа выводит на экран следующие сообщения.

```
Функция vfunc() из класса base.
Функция vfunc() из класса derived1.
Функция vfunc() из класса derived1.
```

Чисто виртуальные функции

Итак, если виртуальная функция не замещается в производном классе, вызывается ее версия из базового класса. Однако во многих случаях невозможно создать разумную версию виртуальной функции в базовом классе. Например, базовый класс может не обладать достаточным объемом информации для создания виртуальной функции. Кроме того, в некоторых ситуациях необходимо гарантировать, что виртуальная функция будет замещена во всех производных классах. Для этих ситуаций в языке C++ предусмотрены чисто виртуальные функции.

Чисто виртуальная функция (pure virtual function) — это виртуальная функция, не имеющая определения в базовом классе. Для объявления чисто виртуальной функции используется следующая синтаксическая конструкция.

```
virtual тип имя_функции(список_параметров) = 0;
```

Чисто виртуальные функции должны переопределяться в каждом производном классе, в противном случае возникнет ошибка компиляции.

Следующая программа содержит простой пример чисто виртуальной функции. Базовый класс **number** содержит целое число **val**, функцию **setval()** и чисто виртуальную функцию **show()**. Производные классы **hextype**, **dectype** и **octtype** являются наследниками класса **number** и переопределяют функцию **show()** так, что она выводит значение **val** в соответствующей системе счисления (шестнадцатеричной, десятичной или восьмеричной).

```
#include <iostream>
using namespace std;

class number {
protected:
  int val;
public:
  void setval(int i) { val = i; }

  // Функция show() является чисто виртуальной.
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show(); // Выводит десятичное число 20.

  h.setval(20);
  h.show(); // Выводит шестнадцатеричное число 14.

  o.setval(20);
  o.show();  // Выводит восьмеричное число 24.

  return 0;
}
```

Несмотря на простоту этого примера, он достаточно ярко иллюстрирует ситуацию, когда в базовом классе невозможно дать осмысленное определение виртуальной функции. В данном случае класс **number** просто обеспечивает единообразный интерфейс для использования производных типов. Функцию **show()** невозможно определить в классе **number**, поскольку в нем не задана основа системы счисления. Однако чисто виртуальная функция **show()** гарантирует, что в каждом производном классе она будет соответствующим образом переопределена.

Следует иметь в виду, что все производные классы обязаны переопределять чисто виртуальную функцию. Если этого не сделать, возникнет ошибка компиляции.

Абстрактные классы

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется *абстрактным* (abstract class). Поскольку абстрактный класс содержит одну или несколько функций, не имеющих определения (т.е. чисто виртуальные функции), его объекты создать невозможно. Следовательно, абстрактные классы можно использовать лишь как основу для производных классов.

Несмотря на то что объекты абстрактного класса не существуют, можно создать указатели и ссылки на абстрактный класс. Это позволяет применять абстрактные классы для поддержки динамического полиморфизма и выбирать соответствующую виртуальную функцию в зависимости от типа указателя или ссылки.

Применение виртуальных функций

В основу объектно-ориентированного программирования положен принцип "один интерфейс, несколько методов". Он позволяет определять базовый класс операций с единообразным интерфейсом, а их конкретизацию предоставлять производным классам.

Виртуальные функции, абстрактные классы и динамический полиморфизм представляют собой один из наиболее мощных и гибких механизмов реализации принципа "один интерфейс, несколько методов". Используя этот механизм, можно создавать иерархии классов, организованные по принципу "от общего — к частному" (от базового класса — к производным). По этому методу в базовом классе следует определять все универсальные свойства и интерфейсы. Если некоторую операцию можно реализовать только в производном классе, используется виртуальная функция. По существу, в базовом классе описываются лишь самые общие свойства, а в производных классах они конкретизируются.

Проиллюстрируем сказанное следующим примером, в котором создается иерархия классов, выполняющих преобразование единицы измерения из одной системы мер в другую (например, литров — в галлоны). В базовом классе **convert** объявлены две переменные — **val1** и **val2**, в которых хранятся исходное и преобразованное значения соответственно. Кроме того, в нем определены функции **getinit()** и **getconv()**, возвращающие эти значения. Эти члены класса **convert** фиксированы и могут использоваться во всех производных классах. Однако функция **compute()**, выполняющая фактическое преобразование, является чисто виртуальной и должна определяться во всех классах, производных от класса **convert**. Конкретный смысл функции **compute()** зависит от типа выполняемого преобразования.

```
// Применение виртуальной функции.
#include <iostream>
using namespace std;

class convert {
protected:
  double val1;  // Начальное значение.
  double val2;  // Преобразованное значение.
public:
  convert(double i) {
    val1 = i;
  }
  double getconv() { return val2; }
  double getinit() { return val1; }

  virtual void compute() = 0;
};

// Преобразование литров в галлоны.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};

// Преобразование шкалы Фаренгейта в шкалу Цельсия.
class f_to_c : public convert {
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p;  // Указатель на базовый класс.

  l_to_g lgob(4);
  f_to_c fcob(70);

  // Применение виртуальной функции.
  p = &lgob;
  cout << p->getinit() << " литров равно ";
  p->compute();
  cout << p->getconv() << " галлонов\n";  // l_to_g

  p = &fcob;
  cout << p->getinit() << " по Фаренгейту равно ";
  p->compute();
  cout << p->getconv() << " по Цельсию\n";  // f_to_c

  return 0;
}
```

В этой программе создаются классы **l_to_g** и **f_to_c**, производные от класса **convert**. Эти классы выполняют преобразования объема, измеренного в литрах, в объем, выраженный в галлонах, а также переводят шкалу температур по Фаренгейту в шкалу по

Цельсию соответственно. Каждый производный класс замещает функцию **compute()** по-своему, выполняя необходимое преобразование. Однако, несмотря на различие фактических преобразований (т.е. методов) в классах **l_to_g** и **f_to_c**, их интерфейс одинаков.

Виртуальные функции позволяют очень легко реагировать на новую ситуацию. Предположим, что в предыдущей программе необходимо предусмотреть преобразование футов в метры, включив его в класс **convert**.

```
// Преобразование футов в метры
class f_to_m : public convert {
public:
  f_to_m(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.28;
  }
};
```

Абстрактные классы и виртуальные функции позволяют создавать *библиотеки классов* (class library), носящие обобщенный характер. Любой программист может создать класс, производный от библиотечного, добавив свои собственные функции. При этом будет сохранен единообразный интерфейс, определенный базовым классом. Таким образом, библиотечные классы можно адаптировать к новым ситуациям.

В заключение отметим, что базовый класс **convert** служит примером абстрактного класса. Виртуальная функция **compute()** не определяется в классе **convert**, поскольку в нем нет информации о выполняемом преобразовании. Функция **compute()** наполняется конкретным смыслом лишь в производных классах.

Сравнение раннего и позднего связывания

Прежде чем завершить главу, посвященную виртуальным функциям и динамическому полиморфизму, следует определить два термина, часто используемых в дискуссиях о языке C++ и объектно-ориентированном программировании: *раннее связывание* (early binding) и *позднее связывание* (late binding).

Раннее связывание означает события, происходящие на этапе компиляции. По существу, раннее связывание означает, что на этапе компиляции известна вся информация, позволяющая выбрать вызываемую функцию. (Иначе говоря, объект и вызов функции связываются друг с другом на этапе компиляции.) Примерами раннего связывания являются обычные вызовы функций (включая стандартные библиотечные функции), вызовы перегруженных функций и операторов. Основное преимущество раннего связывания — эффективность. Поскольку вся информация о вызываемой функции на этапе компиляции уже известна, сам вызов функции происходит очень быстро.

Позднее связывание является антиподом раннего. В языке C++ позднее связывание означает, что фактический выбор вызываемой функции осуществляется только в ходе выполнения программы. Основным средством позднего связывания являются виртуальные функции. Как известно, выбор вызываемой виртуальной функции зависит от типа указателя или ссылки, с помощью которых она вызывается. Поскольку в большинстве случаев на этапе компиляции эта информация отсутствует, связывание объекта и вызова функции откладывается до выполнения программы. Основное преимущество позднего связывания — гибкость. В отличие от раннего, позднее связывание позволяет создавать программу, реагирующую на события, происходящие в ходе ее выполнения, без использования большого количества кода, предусматривающего всевозможные варианты. Однако позднее связывание может замедлить работу программы.

Полный справочник по C++

Глава 18

Шаблоны

Шаблоны — один из наиболее сложных и мощных механизмов языка C++. Их не было в первоначальной версии языка C++, но через несколько лет они стали его неотъемлемой частью, поддерживаемой всеми современными компиляторами. С помощью шаблонов можно создавать обобщенные функции и классы, которые работают с типом данных, заданным как параметр. Таким образом, одну и ту же функцию или класс можно применять к разным типам данным, не используя отдельные варианты для каждого типа.

Обобщенные функции

Обобщенная функция определяет универсальную совокупность операций, применимых к различным типам данных. Тип данных, с которыми работает функция, передается в качестве параметра. Это позволяет применять одну и ту же функцию к широкому спектру данных. Как известно, многие алгоритмы носят универсальный характер и не зависят от типа данных, которыми они оперируют. Например, для массивов целых и действительных чисел используется один алгоритм быстрой сортировки. С помощью обобщенной функции можно определить природу алгоритма независимо от типа данных. После этого компилятор автоматически генерирует правильный код, соответствующий конкретному типу. По существу, обобщенная функция автоматически перегружает саму себя.

Обобщенная функция объявляется с помощью ключевого слова **template**. Обычное значение слова "template" (шаблон) точно соответствует своему смыслу в языке C++. Оно используется для создания шаблона, который описывает действия функции и позволяет компилятору самому уточнять необходимые детали. Определение шаблонной функции выглядит следующим образом.

```
template <class Ттип> тип_возвращаемого_значения
имя_функции(список_параметров)
{
 // Тело функции
}
```

Здесь параметр *Ттип* задает тип данных, с которым работает функция. Этот параметр можно использовать и внутри функции, однако при создании конкретной версии обобщенной функции компилятор автоматически подставит вместо него фактический тип. Традиционно обобщенный тип задается с помощью ключевого слова **class**, хотя вместо него можно применять ключевое слово **typename**.

В следующем примере демонстрируется обобщенная функция, меняющая местами две переменные. Поскольку процесс перестановки не зависит от типа переменных, его можно описать с помощью обобщенной функции.

```
// Пример шаблонной функции.
#include <iostream>
using namespace std;

// Шаблон функции.
template <class X> void swapargs(X &a, X &b)
{
 X temp;

 temp = a;
 a = b;
 b = temp;
}
```

```
int main()
{
 int i=10, j=20;
 double x=10.1, y=23.3;
 char a='x', b='z';

 cout << "Исходные значения i, j: " << i << ' ' << j << '\n';
 cout << "Исходные значения x, y: " << x << ' ' << y << '\n';
 cout << "исходные значения a, b: " << a << ' ' << b << '\n';

 swapargs(i, j); // Перестановка целых чисел.
 swapargs(x, y); // Перестановка действительных чисел.
 swapargs(a, b); // Перестановка символов.

 cout << "переставленные значения i, j: " << i << ' ' << j << '\n';
 cout << "Переставленные значения x, y: " << x << ' ' << y << '\n';
 cout << "Переставленные значения a, b: " << a << ' ' << b << '\n';

 return 0;
}
```

Рассмотрим эту программу подробнее. Строка

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору, что: во-первых, создается шаблон, и, во-вторых, начинается описание обобщенной функции. Здесь параметр **X** задает обобщенный тип, который впоследствии будет заменен фактическим типом. После этой строки объявляется функция **swapargs()**, в которой переменные, подлежащие перестановке, имеют обобщенный тип **X**. В функции **main()** функция **swapargs()** вызывается для трех разных типов данных: **int**, **double** и **char**. Поскольку функция **swapargs()** является обобщенной, компилятор автоматически создает три ее версии: для перестановки целых и действительных чисел, а также символов.

С шаблонами связано несколько понятий. Во-первых, обобщенная функция (т.е. функция, объявленная с помощью ключевого слова **template**) называется также *шаблонной функцией* (template function). Эти термины являются синонимами. Конкретная версия обобщенной функции, создаваемая компилятором, называется *специализацией* (specialization) или *генерируемой функцией* (generated function). Процесс генерации конкретной функции называется *конкретизацией* (instantiation) . Иными словами, генерируемая функция является конкретным экземпляром обобщенной функции.

Поскольку язык C++ не считает конец строки символом конца оператора, раздел **template** в определении обобщенной функции не обязан находиться в одной строке с именем функции. Эту особенность иллюстрирует следующий пример.

```
template <class X>
void swapargs(X &a, X &b)
{
 X temp;

 temp = a;
 a = b;
 b = temp;
}
```

В этом случае следует помнить, что между оператором **template** и началом определения обобщенной функции не должно быть других операторов. Например, следующий фрагмент ошибочен.

```
// Этот фрагмент содержит ошибку.
template <class X>
int i; // Ошибка
void swapargs(X &a, X &b)
{
 X temp;

 temp = a;
 a = b;
 b = temp;
}
```

Как следует из комментария, спецификация **template** должна непосредственно предшествовать определению функции.

Функция с двумя обобщенными типами

Используя список, элементы которого разделены запятыми, можно определить несколько обобщенных типов данных в операторе **template**. Например, в следующей программе создается шаблонная функция, имеющая два обобщенных типа.

```
#include <iostream>
using namespace std;

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
 cout << x << ' ' << y << '\n';
}

int main()
{
 myfunc(10, "Я люблю C++");

 myfunc(98.6, 19L);

 return 0;
}
```

При генерации конкретных экземпляров функции **myfunc()** компилятор заменяет шаблонные типы **type1** и **type2** типами **int** и **char***, а также **double** и **long** соответственно.

Внимание! *При создании шаблонной функции компилятор может автоматически генерировать столько ее различных вариантов, сколько существует способов вызова этой функции в программе.*

Явная перегрузка обобщенной функции

Несмотря на то что обобщенная функция перегружает сама себя, ее можно перегрузить явно. Этот процесс называется *явной специализацией* (explicit specialization). Перегруженная функция замещает (или "маскирует") обобщенную функцию, связанную с данной конкретной версией. Рассмотрим модифицированную версию программы, предназначенной для перестановки двух переменных.

```
// Замещение шаблонной функции.
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
 X temp;

 temp = a;
 a = b;
 b = temp;
 cout << "Внутри функции swapargs.\n";
}

// Эта функция замещает обобщенную версию функции
// swapargs(), предназначенную для перестановки целых чисел. void
swapargs(int &a, int &b)
{
 int temp;

 temp = a;
 a = b;
 b = temp;
 cout << "Внутри специализации функции swapargs для целых чисел.\n";
}

int main()
{
 int i=10, j=20;
 double x=10.1, y=23.3;
 char a='x', b='z';

 cout << "Исходные значения i, j: " << i << ' ' << j << '\n';
 cout << "Исходные значения x, y: " << x << ' ' << y << '\n';
 cout << "Исходные значения a, b: " << a << ' ' << b << '\n';

 swapargs(i, j); // Вызов явно перегруженной функции swapargs().
 swapargs(x, y); // Вызов обобщенной функции swapargs().
 swapargs(a, b); // Вызов обобщенной функции swapargs()

 cout << "Переставленные значения i, j: " << i << ' ' << j << '\n';
 cout << "Переставленные значения x, y: " << x << ' ' << y << '\n';
 cout << "Переставленные значения a, b: " << a << ' ' << b << '\n';

 return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Исходные значения i, j: 10 20
Исходные значения x, y: 10.1 23.3
Исходные значения a, b: x z
Внутри специализации функции swapargs для целых чисел.
Внутри шаблонной функции swapargs.
Внутри шаблонной функции swapargs.
Переставленные значения i, j: 20 10
Переставленные значения x, y: 23.3 10.1
Переставленные значения a, b: z x
```

Как указано в комментариях программы, при вызове функции **swapargs()** активизируется ее явно перегруженная версия. Таким образом, компилятор не генерирует версию шаблонной функции **swapargs()**.

Недавно появилась новая синтаксическая конструкция, предназначенная для обозначения явной специализации функции. Этот метод использует ключевое слово **template**. Например, перегруженную функцию **swapargs()** из предыдущего примера можно переписать следующим образом.

```
// Новая синтаксическая конструкция для специализации.
template<> void swapargs<int>(int &a, int &b)
{
 int temp;

 temp = a;
 a = b;
 b = temp;
 cout << "Внутри специализации функции swapargs для целых чисел.\n";
}
```

Как видим, новый способ определения специализации содержит конструкцию **template<>**. Тип данных, для которого предназначена специализация, указывается внутри угловых скобок после имени функции. Для специализации любого другого типа обобщенной функции используется такая же синтаксическая конструкция. В настоящее время оба способа определения специализации эквивалентны. Вероятно, в будущем новый стиль может оказаться более предпочтительным.

С помощью явной специализации шаблона можно настраивать версию обобщенной функции на конкретную ситуацию. Это позволяет максимально эффективно использовать ее для заданного типа данных. Однако, как правило, если для разных типов данных требуются разные версии функции, следует применять перегрузку, а не шаблоны.

Перегрузка шаблонной функции

Для того чтобы перегрузить спецификацию обобщенной функции, достаточно создать еще одну версию шаблона, отличающуюся от остальных своим списком параметров.

```
// Перегрузка шаблонной функции.
#include <iostream>
using namespace std;

// Первая версия шаблонной функции f()
template <class X> void f(X a)
{
 cout << "Внутри функции f(X a)\n";
}

// Первая версия шаблонной функции f().
template <class X, class Y> void f(X a, Y b)
{
 cout << "Внутри функции f(X a, Y b)\n";
}

int main()
{
 f(10); // Вызов функции f(X).
 f(10, 20); // Вызов функции f(X, Y).
```

```
 return 0;
}
```

Здесь шаблон функции **f()** перегружен для одного и двух параметров.

Использование стандартных параметров шаблонных функций

При определении шаблонной функции можно смешивать стандартные и обобщенные параметры. В этом случае стандартные параметры ничем не отличаются от параметров любых других функций. Рассмотрим пример.

```
// Применение стандартных параметров в шаблонной функции.
#include <iostream>
using namespace std;

const int TABWIDTH = 8;

// Выводит на экран данные в позиции tab.
template<class X> void tabOut(X data, int tab)
{
 for(; tab; tab--)
 for(int i=0; i<TABWIDTH; i++) cout << ' ';

 cout << data << "\n";
}

int main()
{
 tabOut("Проверка", 0);
 tabOut(100, 1);
 tabOut('X', 2);
 tabOut(10/3, 3);

 return 0;
}
```

Программа выводит на экран следующие сообщения.

```
Проверка
 100
 X
 3
```

В этой программе функция **tabOut()** выводит на экран свой первый аргумент, позиция которого определяется вторым параметром **tab**. Поскольку первый аргумент имеет обобщенный тип, функция **tabOut()** может выводить на экран данные любого типа. Параметр **tab** является стандартным и передается по значению. Смешение обобщенных и стандартных параметров не вызывает никаких проблем и может оказаться полезным.

Ограничения на обобщенные функции

Обобщенные функции напоминают перегруженные, но на них налагаются еще более жесткие ограничения. При перегрузке внутри тела каждой функции можно выполнять разные операции. В то же время обобщенная функция должна выполнять одну и ту же универсальную операцию для всех версий, различаться могут лишь типы данных. Рассмотрим перегруженную функцию на следующем примере. Эти функции *нельзя* заменять обобщенными, поскольку они имеют разное предназначение.

```
#include <iostream>
#include <cmath>
using namespace std;

void myfunc(int i)
{
 cout << "Значение = " << i << "\n";
}

void myfunc(double d)
{
 double intpart;
 double fracpart;

 fracpart = modf(d, &intpart);
 cout << "Дробная часть = " << fracpart;
 cout << "\n";
 cout << "Целая часть = " << intpart;
}

int main()
{
 myfunc(1);
 myfunc(12.2);

 return 0;
}
```

Применение обобщенных функций

Возможность использовать обобщенные функции — одно из самых ценных свойств языка C++. Их можно применять в самых разных ситуациях. Как указывалось ранее, если функция описывает универсальный алгоритм, ее можно сделать шаблонной. После этого ее можно применять к любому типу данных. Прежде чем перейти к описанию шаблонных классов, рассмотрим два примера обобщенных функций, демонстрирующих эффективность и удобство этого механизма.

Обобщенная сортировка

Сортировка представляет собой типичный пример универсального алгоритма. Как правило, алгоритм сортировки совершенно не зависит от типа сортируемых данных. Следующая программа демонстрирует обобщенную функцию, предназначенную для сортировки данных методом пузырька. Несмотря на то что этот алгоритм сортировки является одним из самых медленных, он очень прост и нагляден. Функция **bubble()** упорядочивает массив любого типа. Ей передается указатель на первый элемент массива и количество элементов в массиве.

```
// Обобщенная сортировка методом пузырька.
#include <iostream>
using namespace std;
template <class X> void bubble(
   X *items, // Указатель на упорядочиваемый массив.
   int count) // Количество элементов массива.
{
   register int a, b;
```

```
    X t;
    for(a=1; a<count; a++)
      for(b=count-1; b>=a; b--)
      if(items[b-1] > items[b]) {
      // Перестановка элементов.
      t = items[b-1];
      items[b-1] = items[b];
      items[b] = t;
    }
}

int main()
{
  int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
  double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

  int i;

  cout << "Неупорядоченный массив целых чисел: ";
  for(i=0; i<7; i++)
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Неупорядоченный массив действительных чисел: ";
  for(i=0; i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  bubble(iarray, 7);
  bubble(darray, 5);

  cout << "Упорядоченный массив целых чисел: ";
  for(i=0; i<7; i++)
    cout << iarray[i] << ' ';
  cout << endl;

  cout << "Упорядоченный массив действительных чисел: ";
  for(i=0; i<5; i++)
    cout << darray[i] << ' ';
  cout << endl;

  return 0;
}
```

Программа выводит на экран следующие результаты.

```
Неупорядоченный массив целых чисел: 7 5 4 3 9 8 6
Неупорядоченный массив действительных чисел: 4.3 2.5 -0.9 100.2 3
Упорядоченный массив целых чисел: 3 4 5 6 7 8 9
Упорядоченный массив действительных чисел: -0.9 2.5 3 4.3 100.2
```

Как видим, в программе создаются два массива, имеющие соответственно типы **int** и **double**. Поскольку функция **bubble()** является шаблонной, она автоматически перегружается для каждого из этих типов. Эту функцию можно применить для сортировки данных другого типа, даже объектов какого-нибудь класса. Компилятор автоматически создаст соответствующую версию функции для нового типа.

Уплотнение массива

Проиллюстрируем преимущества шаблонов на примере функции **compact()**. Эта функция уплотняет элементы массива. Довольно часто приходится удалять несколько элементов из середины массива и перемещать оставшиеся элементы влево, заполняя образовавшуюся пустоту. Эта операция носит универсальный характер и не зависит от типа массива. Обобщенная функция **compact()** получает указатель на первый элемент массива, количество его элементов, а также индексы первого и последнего элементов удаленного отрезка массива. Затем функция удаляет указанные элементы и уплотняет массив. Для иллюстрации неиспользуемые элементы, образующиеся при уплотнении массива, обнуляются.

```
// Обобщенная функция уплотнения массива.
#include <iostream>
using namespace std;

template <class X> void compact(
  X *items,  // Указатель на уплотняемый массив.
  int count, // Количество элементов массива.
  int start, // Индекс первого удаленного элемента.
  int end) // Индекс последнего удаленного элемента.
{
  register int i;

  for(i=end+1; i<count; i++, start++)
    items[start] = items[i];

 /* Для иллюстрации оставшаяся часть массива
    заполняется нулями. */
  for( ; start<count; start++) items[start] = (X) 0;
}

int main()
{
  int nums[7] = {0, 1, 2, 3, 4, 5, 6};
  char str[18] = "Обобщенные функции";

  int i;

  cout << "Неуплотненная часть целочисленного массива: ";
  for(i=0; i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Неуплотненная часть строки: ";
  for(i=0; i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  compact(nums, 7, 2, 4);
  compact(str, 18, 6, 10);

  cout << "Уплотненный целочисленный массив: ";
  for(i=0; i<7; i++)
    cout << nums[i] << ' ';
  cout << endl;

  cout << "Уплотненная целочисленная строка: ";
```

```
  for(i=0; i<18; i++)
    cout << str[i] << ' ';
  cout << endl;

  return 0;
}
```

Эта программа уплотняет массивы двух типов: целочисленный и строку. Однако функция **compact()** может работать с массивом любого типа. Результаты работы программы показаны ниже.

```
Неуплотненный целочисленный массив: 0 1 2 3 4 5 6
Неуплотненный целочисленный массив: О б о б щ е н н а я  Ф у н к ц и я
Уплотненный целочисленный массив: 0 1 5 6 0 0 0
Уплотненная строка: О б о б щ е н н а я ц и я
```

Как видим, во многих случаях шаблоны являются вполне естественными. Если логика функции не зависит от типа данных, ее можно преобразовать в шаблонную.

Обобщенные классы

Кроме обобщенных функций можно определить обобщенные классы. При этом создается класс, в котором определены все алгоритмы, однако фактический тип данных задается в качестве параметра при создании объекта.

Обобщенные классы оказываются полезными, если логика класса не зависит от типа данных. Например, к очередям, состоящим из целых чисел или символов, можно применять один и тот же алгоритм, а для поддержки связанных списков адресов рассылки и мест для парковки автомобилей использовать один и тот же механизм.

Объявление обобщенного класса имеет следующий вид.

```
template <class Ttип> class имя_класса
{
  .
  .
  .
}
```

Здесь параметр *Ttип* задает тип данных, который уточняется при создании экземпляра класса. При необходимости можно определить несколько обобщенных типов, используя список имен, разделенных запятой.

Конкретный экземпляр обобщенного класса создается с помощью следующей синтаксической конструкции.

```
имя_класса <тип> имя_объекта;
```

Здесь параметр *тип* задает тип данных, которыми оперирует класс. Функции — члены обобщенного класса автоматически становятся обобщенными. Для их объявления не обязательно использовать ключевое слово **template**.

Следующая программа использует обобщенный класс **stack** (см. главу 11). Теперь его можно применять для хранения объектов любого типа. В данном примере создаются стеки символов и действительных чисел.

```
// Демонстрация обобщенного стека.
#include <iostream>
using namespace std;

const int SIZE = 10;
```

```
// Создаем обобщенный класс stack.
template <class StackType> class stack
{
  StackType stck[SIZE]; // Содержит элементы стека.
  int tos; // Индекс вершины стека.

public:
  stack() { tos = 0; } // Инициализирует стек.
  void push(StackType ob); // Заталкивает объект в стек.
  StackType pop(); // Выталкивает объект из стека.
};

  // Заталкиваем объект в стек.
  template <class StackType>
  void stack<StackType>::push(StackType ob)
  {
    if(tos==SIZE) {
    cout << "Стек полон.\n";
    return;
    }
    stck[tos] = ob;
    tos++;
  }

// Выталкиваем объект из стека.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
  cout << "Стек пуст..\n";
  return 0; // Если стек пуст, возвращается константа null.
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Демонстрация стека символов.
  stack<char> s1, s2; // Создаем два стека символов.
  int i;
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Выталкиваем s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Выталкиваем s2: " << s2.pop() << "\n";

  // Демонстрация стека действительных чисел
  stack<double> ds1, ds2; // Создаем два стека
                          // действительных чисел.
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
```

```
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Выталкиваем ds1: " << ds1.pop() <<
"\n";
  for(i=0; i<3; i++) cout << "Выталкиваем ds2: " << ds2.pop() <<
"\n";

  return 0;
}
```

Как видим, объявление обобщенного класса мало отличается от объявления обобщенной функции. Фактический тип данных, размещаемый в стеке, в объявлении класса заменяется обобщенным параметром и уточняется лишь при создании конкретного объекта. При объявлении конкретного объекта класса **stack** компилятор автоматически генерирует все функции и переменные, необходимые для обработки фактических данных. В предыдущем примере объявляются по два стека разных типов — целых чисел и действительных чисел. Обратите особое внимание на следующие объявления.

```
stack<char> s1, s2; // Создаем два стека символов.
stack<double> ds1, ds2; // Создаем два стека действительных чисел.
```

Как видим, требуемый тип данных задается в угловых скобках. Изменяя этот тип при создании объекта класса **stack**, можно изменять тип данных, хранящихся в стеке. Например, используя следующее определение, можно создать другой стек для хранения указателей на символы.

```
stack<char *> chrptrQ;
```

Можно создавать стеки, хранящие объекты, тип которых определен пользователем. Допустим, что для хранения информации используется следующая структура.

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
};
```

В этом случае класс **stack** порождает стек, в котором хранятся объекты класса **add**. Для этого используется следующее объявление.

```
stack<addr> obj;
```

Класс **stack** демонстрирует, что обобщенные функции и классы являются мощным средством, облегчающим программирование. С его помощью программист может определять общую форму объекта, в котором хранятся данные произвольного типа, и не заботиться об отдельных реализациях классов и функций, предназначенных для разных типов. Компилятор автоматически создает конкретные версии класса.

Пример использования двух обобщенных типов данных

Шаблонный класс может иметь несколько обобщенных типов. Для этого их следует перечислить в списке шаблонных параметров в объявлении **template**. Например, следующая программа создает класс, использующий два обобщенных типа.

```
/*  Пример класса, использующего два обобщенных типа  */

#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

  int main()
  {
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Шаблоны — мощный механизм.");

    ob1.show(); // Выводим целое и действительное число.
    ob2.show(); // Выводим символ и указатель на символ.

    return 0;
}
```

Эта программа выводит следующие результаты.

```
10 0.23
X Шаблоны — мощный механизм.
```

В программе объявляются объекты двух типов. Объект **ob1** использует целые и действительные числа. Объект **ob2** использует символ и указатель на символ. В обоих случаях при создании объектов компилятор автоматически генерирует соответствующие данные и функции.

Применение шаблонных классов: обобщенный массив

Чтобы проиллюстрировать практические выгоды, которые предоставляют обобщенные классы, рассмотрим способ, который довольно часто применяется. Как указано в главе 15, оператор "`[]`" можно перегрузить. Это позволяет создавать собственные реализации массива, в том числе "безопасные" массивы, предусматривающие проверку диапазона индексов в ходе выполнения программы. Как известно, в языке C++ нет встроенной проверки диапазона индексов, поэтому в ходе выполнения программы индекс может выйти за допустимые пределы, не генерируя сообщения об ошибке. Однако, если создать класс, содержащий массив, и перегрузить оператор "`[]`", выход индекса за допустимые пределы можно предотвратить.

Комбинируя перегруженный оператор с шаблонным классом, можно создать обобщенный безопасный массив произвольного типа. Этот тип массива показан в следующей программе.

```
// Пример обобщенного безопасного массива.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;
```

```
template <class AType> class atype {
  AType a[SIZE];
public:
  atype() {
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Проверка диапазона для объекта atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
  cout << "\nЗначение индекса  ";
  cout << i << " выходит за пределы допустимого диапазона.\n";
  exit(1);
  }
  return a[i];
}

int main()
{
  atype<int> intob;        // Целочисленный массив.
  atype<double> doubleob; // Массив действительных чисел.

  int i;

  cout << "Целочисленный массив: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "массив действительных чисел: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // Генерирует сообщение об ошибке.

  return 0;
}
```

Эта программа реализует обобщенный тип массива и демонстрирует его применение на примере массивов целых и действительных чисел. Попробуйте создать массивы других типов. Как показывает данный пример, обобщенные классы позволяют создавать один вариант кода, отлаживать его, а затем применять к любому типу данных, не предусматривая для каждого типа свой вариант.

Применение стандартных типов в обобщенных классах

В спецификации шаблона обобщенного класса можно использовать стандартные типы. Иначе говоря, в качестве шаблонных параметров можно применять стандартные аргументы, например, целочисленные значения или указатели. Синтаксис такого объявления не отличается от объявления обычных параметров функций: необходимо лишь указать тип и имя аргумента. Рассмотрим один из наиболее удачных способов реализации безопасных обобщенных массивов, позволяющий задавать размер массива.

```
// Демонстрация стандартных шаблонных параметров.
#include <iostream>
#include <cstdlib>
using namespace std;

// Здесь целочисленный аргумент size является стандартным.
template <class AType, int size> class atype {
  AType a[size]; // Длина массива передается через параметр size
public:
  atype()
  {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Проверка диапазона для объекта atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
  cout << "\nЗначение индекса ";
  cout << i << " выходит за пределы допустимого диапазона.\n";
  exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 10> intob; // Целочисленный массив из 10 элементов.
  atype<double, 15> doubleob; // Массив действительных чисел,
                              // состоящий из 16 элементов.
  int i;

  cout << "Целочисленный массив: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "массив действительных чисел: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // Генерирует сообщение об ошибке.

  return 0;
}
```

Внимательно рассмотрим спецификацию шаблона **atype**. Обратите внимание на то, что параметр **size** объявлен как целочисленный. Затем этот параметр используется в классе **atype** для объявления массива **a**. Хотя параметр **size** применяется в исходном коде как обычная переменная, его значение известно уже на этапе компиляции. Это позволяет задавать размер массива. Кроме того, параметр **size** используется при проверке диапазона индекса в операторной функции **operator[]()**. Обратите

внимание на способ, которым создаются массивы целых и действительных чисел. Второй параметр задает размер каждого массива.

В качестве стандартных параметров можно применять лишь целые числа, указатели и ссылки. Другие типы, например **float**, не допускаются. Аргументы, которые передаются стандартным параметрам, должны содержать либо целочисленную константу, либо указатель, либо ссылку на глобальную функцию или объект. Таким образом, стандартные параметры можно рассматривать как константы, поскольку их значения нельзя изменять. Например, внутри функции **operator[]()** следующий оператор не допускается.

```
size = 10; // Ошибка
```

Поскольку стандартные параметры считаются константами, с их помощью можно задавать размер массива, что довольно важно в практических приложениях.

Как показывает пример безопасного обобщенного массива, стандартные параметры значительно расширяют возможности обобщенных классов. Хотя значения стандартных аргументов должны быть известны уже на этапе компиляции, это ограничение не слишком обременительно по сравнению с мощностью, предоставляемой ими

Применение аргументов по умолчанию в шаблонных классах

Шаблонный класс может иметь аргумент обобщенного типа, значение которого задано по умолчанию. Например, такой.

```
template <class X=int> class myclass { //...
```

Если при конкретизации объекта типа **myclass** не будет указан ни один тип, используется тип **int**.

Стандартные аргументы также могут иметь значения по умолчанию. Они используются при конкретизации объекта, если не заданы явные значения аргументов. Синтаксическая конструкция, применяемая для этих параметров, не отличается от объявления функций, имеющих аргументы по умолчанию.

Рассмотрим еще один вариант безопасного массива, предусматривающий аргументы по умолчанию как для типа данных, так и для размера массива.

```
// Демонстрация шаблонных аргументов по умолчанию.
#include <iostream>
#include <cstdlib>
using namespace std;

// Параметр типа AType по умолчанию равен int,
// а переменная size по умолчанию равна 10.

template <class AType=int, int size=10> class atype {
  AType a[size]; // Размер массива передается аргументом size.
public:
  atype()
  {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Проверка диапазона для объекта atype.
template <class AType, int size>
  AType &atype<AType, size>::operator[](int i)
{
```

```
  if(i<0 || i> size-1) {
  cout << "\nЗначение индекса ";
  cout << i << " выходит за пределы допустимого диапазона.\n";
  exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 100> intarray;  // Целочисленный массив
                             // из 100 элементов.
  atype<double> doublearray; // Массив действительных чисел,
                             // размер задан по умолчанию.
  atype<> defarray; // По умолчанию объявляется целочисленный
                    // массив, состоящий из 10 элементов.

  int i;

  cout << "Целочисленный массив: ";
  for(i=0; i<100; i++) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << " ";
  cout << '\n';

  cout << "Массив действительных чисел: ";
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << " ";
  cout << '\n';

  cout << "Массив по умолчанию: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << " ";
  cout << '\n';

  return 0;
}
```

Обратите внимание на строку

```
template <class AType=int, int size=10> class atype {
```

Здесь тип **AType** по умолчанию является типом **int**, а переменная **size** равна 10. Как демонстрирует программа, объекты класса **atype** можно создать тремя способами.

- Явно задавая тип и размер массива.
- Явно задавая тип массива, используя размер по умолчанию.
- Используя тип и размер массива, установленные по умолчанию.

Применение аргументов по умолчанию (особенно типов) повышает универсальность шаблонных классов. Если некий тип используется чаще других, его можно задать по умолчанию, предоставив пользователю самому конкретизировать другие типы.

Явные специализации классов

Как и при использовании шаблонных функций, можно создать явную специализацию обобщенного класса. Для этого, как и прежде, применяется конструкция **template<>**.

```
// Демонстрация специализации класса.
#include <iostream>
```

```
using namespace std;

template <class T> class myclass {
  T x;
public:
  myclass(T a) {
  cout << "Внутри обобщенного класса myclass\n";
  x = a;
  }
  T getx() { return x; }
};

// Явная специализация для типа int.
template <> class myclass<int> {
  int x;
  public:
    myclass(int a) {
    cout << "Внутри специализации myclass<int>\n";
    x = a * a;
    }

  int getx() { return x; }
};

int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";

  return 0;
}
```

Программа выводит на экран следующие результаты.

```
Внутри обобщенного класса myclass
double: 10.1
```

Обратите внимание на следующую строку программы.

```
template <> class myclass<int> {
```

Она сообщает компилятору, что создается явная целочисленная специализация обобщенного класса **myclass**. Такая же синтаксическая конструкция применяется для любой другой специализации класса.

Явная специализация класса расширяет возможности обобщенных классов, поскольку она позволяет обрабатывать одну или две особые ситуации, предоставляя компилятору автоматически генерировать остальные специализации. Если программе потребуется слишком много специализаций, возможно, следует вообще отказаться от обобщенных классов.

Ключевые слова typename и export

Сравнительно недавно в язык C++ были включены ключевые слова, связанные с шаблонами: **typename** и **export**, имеющие особое значение для программирования. Кратко рассмотрим каждое из них.

Ключевое слово **typename** используется в двух ситуациях. Во-первых, как указывалось ранее, оно может заменять ключевое слово **class** в объявлении шаблона. Например, шаблонную функцию **swapargs()** можно определить так.

```
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Здесь ключевое слово **typename** задает обобщенный тип **X**. В этом контексте ключевые слова **class** и **typename** не различаются.

Во-вторых, ключевое слово **typename** информирует компилятор о том, что некое имя используется в объявлении шаблонного класса в качестве имени типа, а не объекта. Рассмотрим пример.

```
typename X::Name someObject;
```

Здесь имя **X::Name** используется в качестве имени типа.

Ключевое слово **export** может предшествовать объявлению **template**. Оно позволяет использовать шаблон из другого файла, повторяя лишь его объявление, а не все определение.

Мощь шаблонов

Шаблоны позволяют решить одну из самых трудных задач программирования: создать код, пригодный к повторному выполнению. С помощью шаблонов можно создавать схемы, применяемые в самых разных ситуациях. Рассмотрим в качестве примера класс **stack**. Стек, описанный в главе 11, можно было использовать лишь для хранения целых чисел. Даже если алгоритм не зависит от типа данных, жестко заданный тип стека ограничивает его применение. Однако обобщенный класс **stack** позволяет создавать стеки любых типов.

Обобщенные функции и классы обеспечивают мощный механизм, позволяющий упростить программирование. Написав и отладив шаблонный класс, вы получаете законченный программный компонент, который можно применять в разных ситуациях. Теперь программист избавлен от необходимости предусматривать отдельные реализации для разных типов данных.

Хотя синтаксис шаблонов, на первый взгляд, выглядит устрашающе, они позволяют сэкономить время и облегчают программирование. Шаблонные функции и классы получили очень широкое применение, и их популярность продолжает расти. Например, библиотека STL (Standard Template Library — стандартная библиотека шаблонов), являющаяся частью стандарта языка C++, как следует из ее названия, построена на основе шаблонов. И последнее: хотя шаблоны образуют дополнительный уровень абстракции, в конце концов они компилируются в высокоэффективный объектный код.

Полный справочник по C++

Глава 19

Обработка исключительных ситуаций

Обработка исключительных ситуаций (exception handling) позволяет правильно реагировать на ошибки, возникающие в ходе выполнения программы. Используя механизм исключительных ситуаций, программа может автоматически вызывать процедуру обработки ошибок. Это избавляет программиста от утомительного кодирования.

Основы обработки исключительных ситуаций

Механизм обработки исключительных ситуаций в языке C++ основан на трех ключевых словах: **try**, **catch** и **throw**. Фрагменты программы, подлежащие контролю, содержат блок **try**. Если в ходе выполнения программы в блоке **try** возникает исключительная ситуация (т.е. ошибка), она генерируется с помощью ключевого слова **throw**. Затем исключительная ситуация перехватывается блоком **catch** и обрабатывается. Уточним это описание.

Код, подлежащий контролю, должен выполняться внутри блока **try**. (Функции, вызываемые внутри блока **try**, также могут генерировать исключительные ситуации.) Исключительные ситуации перехватываются оператором **catch**, который следует непосредственно за блоком **try**, в котором они возникли. Общий вид операторов **try** и **catch** показан ниже.

```
try {
   // Тело блока try
} catch(тип1 аргумент) {
   // Тело блока catch
}
catch(тип2 аргумент) {
   // Тело блока catch
}
catch(тип3 аргумент) {
   // Тело блока catch
}
.
.
.
catch(типN аргумент) {
   // Тело блока catch
}
```

Размер блока **try** может варьироваться. Он может содержать как несколько операторов, так и целую программу (в этом случае функция **main()** целиком помещается в блок **try**).

Возникшая исключительная ситуация перехватывается соответствующим оператором **catch**, который выполняет ее обработку. С одним блоком **try** может быть связано несколько операторов **catch**. Выбор правильного оператора **catch** определяется типом исключительной ситуации. Из нескольких вариантов выбирается оператор **catch**, тип аргумента которого совпадает с возникшей исключительной ситуацией (остальные варианты игнорируются). В процессе перехвата исключительной ситуации *аргументу* присваивается некое значение. Аргумент может быть объектом встроенного типа либо класса. Если фрагмент не генерирует никаких исключительных ситуаций (т.е. в блоке **try** ошибки не возникают), не выполняется ни один оператор **catch**.

Оператор **throw** имеет следующий вид.

```
throw исключительная_ситуация;
```

Оператор **throw** генерирует указанную *исключительную ситуацию*. Если в программе предусмотрен ее перехват, оператор **throw** должен выполняться либо внутри блока **try**, либо внутри функции, явно или неявно вызываемой внутри блока **try**.

Если генерируется исключительная ситуация, для которой не предусмотрена обработка, программа может прекратить свое выполнение. В этом случае вызывается стандартная функция **terminate()**, которая по умолчанию вызывает функцию **abort()**. Однако, как будет показано ниже, программист может предусмотреть собственную обработку ошибки.

Рассмотрим пример, демонстрирующий обработку исключительной ситуации.

```
// Простой пример обработки исключительной ситуации.
#include <iostream>
using namespace std;

int main()
{
  cout << "Начало\n";

  try { // Начало блока try
    cout << "Внутри блока try\n";
    throw 100; // Генерируем ошибку.
    cout << "Этот оператор не выполняется.";
  }  catch (int i) { // Перехват ошибки.
    cout << "перехват исключительной ситуации - значение равно: ";
    cout << i << "\n";
  }

  cout << "Конец";

  return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Начало
Внутри блока try
Перехват исключительной ситуации - значение равно: 100
Конец
```

Внимательно проанализируйте эту программу. Как видите, блок **try** содержит три оператора. С ним связан оператор **catch(int i)**, выполняющий обработку целочисленной исключительной ситуации. Внутри блока **try** выполняются только два из трех операторов: первый оператор **cout** и оператор **throw**. При генерации исключительной ситуации управление передается оператору **catch**, а выполнение блока **try** прекращается. Иначе говоря, блок **catch** *не вызывается*. Просто программа переходит к его выполнению. (Для этого стек программы автоматически обновляется.) Таким образом, оператор **cout**, следующий за оператором **throw**, никогда не выполняется.

Обычно оператор **catch** пытается исправить ошибку, предпринимая соответствующие действия. Если это возможно, выполнение программы возобновляется с оператора, следующего за блоком **catch**. Однако часто ошибку исправить невозможно, и блок **catch** прекращает выполнение программы, вызывая функцию **exit()** или **abort()**.

Как указывалось ранее, тип исключительной ситуации должен совпадать с типом, указанным в операторе **catch**. Например, если в предыдущей программе изменить тип аргумента оператора **catch** на **double**, перехват исключительной ситуации не состоится, и программа завершится аварийно. Проиллюстрируем это следующим примером.

```
// Эта программа не работает
#include <iostream>
```

```
using namespace std;

int main()
{
  cout << "Начало\n";

  try { // Начало блока try.
    cout << "Внутри блока try\n";
    throw 100; // Генерируем ошибку.
    cout << "Этот оператор не выполняется";
  }  catch (double i)  { // Не перехватывает целочисленные
                         // исключительные ситуации.
    cout << "Перехват исключительной ситуации - значение равно: ";
    cout << i << "\n";
  }

  cout << "Конец";

  return 0;
}
```

В этой программе оператор **catch(double i)** не перехватывает целочисленные исключительные ситуации. (Разумеется, точное содержание сообщения, описывающего аварийное завершение программы, зависит от компилятора.) На экран выводятся следующие сообщения.

```
Начало
Внутри блока try
Abnormal program termination
```

Исключение может генерироваться вне блока **try** только в том случае, если оно генерируется функцией, которая вызывается внутри этого блока. Рассмотрим пример.

```
/* Генерирование исключительной ситуации внутри функции,
   находящейся вне блока try.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Внутри функции Xtest, test =: " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "Начало\n";

  try { // Начало блока try.
    cout << "Внутри блока try\n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  } catch (int i) { // Перехват ошибки.
    cout << "Перехват исключительной ситуации - значение равно: ";
    cout << i << "\n";
  }
```

```
  cout << "Конец";

  return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Начало
Внутри блока try
Внутри функции Xtest, test = 0
Внутри функции Xtest, test = 1
Перехват исключительной ситуации - значение равно: 1
Конец
```

Блок **try** может находиться внутри функции. В этом случае при каждом входе в функцию обработка исключительной ситуации выполняется заново. В качестве примера проанализируем следующую программу.

```
#include <iostream>
using namespace std;

// Блоки try/catch находятся внутри функции.
void Xhandler(int test)
{
  try {
    if(test) throw test;
  } catch(int i) {
    cout << "Перехват исключительной ситуации #: " << i << '\n';
  }
}

int main()
{
  cout << "Начало\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "Конец";

  return 0;
}
```

Эта программа выводит на экран следующие сообщения.

```
Начало
Перехват исключительной ситуации #: 1
Перехват исключительной ситуации #: 2
Перехват исключительной ситуации #: 3
Конец
```

Как видим, в ходе выполнения программы были перехвачены три исключительные ситуации. После каждой обработки функция возвращает управление вызывающему модулю. При повторном вызове функции обработка исключительной ситуации выполняется вновь.

Важно четко понимать, что код, связанный с оператором **catch**, выполняется только при перехвате исключительной ситуации. В противном случае оператор **catch** просто игнорируется. (Иначе говоря, поток управления никогда не проходит через те-

ло оператора **catch**.) Например, в следующей программе исключительные ситуации вообще не генерируются, и оператор **catch** не выполняется.

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Начало\n";

  try { // Начало блока try.
    cout << "Внутри блока try\n";
    cout << "Все еще внутри блока try\n";
  } catch (int i) { // Перехват ошибки
    cout << "Перехват исключительной ситуации - значение равно: ";
    cout << i << "\n";
  }

  cout << "Конец";

  return 0;
}
```

Эта программа выводит на экран следующие сообщения.

```
Начало
Внутри блока try
Все еще внутри блока try
Конец
```

Как видим, поток управления обошел оператор **catch** стороной.

Перехват классов исключительных ситуаций

Исключительная ситуация может иметь любой тип, в том числе быть объектом класса, определенного пользователем. В практических приложениях исключительные ситуации, определенные пользователем, встречаются чаще, чем встроенные. Возможно, это происходит потому, что программисты стремятся как можно точнее определять классы исключительных ситуаций и создавать объекты, описывающие вероятные ошибки. Эта информация позволяет обработчику исправлять возникшие ошибки. Проиллюстрируем сказанное следующим примером.

```
// Перехват класса исключительных ситуаций.
#include <iostream>
#include <cstring>
using namespace std;

class MyException {
public:
  char str_what[80];
  int what;

  MyException() { *str_what = 0; what = 0; }

  MyException(char *s, int e) {
    strcpy(str_what, s);
    what = e;
  }
};
```

```
int main()
{
  int i;

  try {
    cout << "Введите положительное число: ";
    cin >> i;
    if(i<0)
      throw MyException("Число не положительно", i);
  } catch (MyException e) { // Перехват ошибки
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }

  return 0;
}
```

В результате на экране появятся следующие строки.

```
Введите положительное число: -4
Число не положительно: -4
```

Программа предлагает пользователю ввести положительное число. Если в ответ пользователь введет отрицательное число, создается объект класса **MyException**, описывающий возникшую ошибку. Таким образом, класс **MyException** инкапсулирует информацию об ошибке. Затем эта информация используется для обработки ошибки Как правило, инкапсуляция информации об ошибке в отдельном классе повышает эффективность обработки исключительных ситуаций.

Применение нескольких операторов catch

Как известно, с одним оператором **try** можно связывать несколько операторов **catch**. Именно такой вариант используется чаще всего. Однако каждый оператор **catch** должен перехватывать отдельный тип исключительной ситуации. Например, следующая программа перехватывает целые числа и строки.

```
#include <iostream>
using namespace std;

// Перехватываются несколько типов исключительных ситуаций.
void Xhandler(int test)
{
  try {
    if(test) throw test;
    else throw "Значение равно нулю";
  }
    catch(int i) {
      cout << "Перехват исключительной ситуации #: " << i << '\n';
  }
    catch(const char *str) {
      cout << "Перехват строки: ";
      cout << str << '\n';
  }
}

int main()
{
```

```
  cout << "Начало\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "Конец";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Начало
Перехват исключительной ситуации #: 1
Перехват исключительной ситуации #: 2
Перехват строки: Значение равно нулю.
Перехват исключительной ситуации #: 3
Конец
```

Как видим, каждый оператор **catch** соответствует отдельному типу.

Как правило, операторы **catch** проверяются в порядке их следования в программе. Выполняется лишь тот оператор, тип аргумента которого точно соответствует возникшей исключительной ситуации. Все остальные блоки **catch** игнорируются.

Обработка производных исключительных ситуаций

Если исключительные ситуации описываются с помощью базового и производных классов, при работе с операторами **catch** следует проявлять максимальную осторожность, поскольку оператор **catch**, соответствующий базовому классу, одновременно соответствует всем производным классам. Таким образом, если необходимо перехватить исключительные ситуации базового и производных классов, в последовательности операторов **catch** производный класс следует обрабатывать первым. Если этого не сделать, оператор **catch**, соответствующий базовому классу исключительной ситуации, также будет перехватывать исключительные ситуации всех производных классов. Рассмотрим следующую программу.

```
// Перехват производных классов.
#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
  D derived;

  try {
```

```
    throw derived;
  }
  catch(B b) {
    cout << "Перехват базового класса.\n";
  }
  catch(D d) {
    cout << "Этот оператор не выполняется.\n";
  }

  return 0;
}
```

Поскольку объект **derived** является экземпляром класса, производного от класса **B**, он будет перехвачен первым оператором **catch**, а второй оператор **catch** никогда выполняться не будет. Некоторые компиляторы в таких случаях выдают предупреждение. Другие компиляторы вообще считают это ошибкой. Так или иначе, чтобы исправить эту ситуацию, следует поменять порядок следования операторов **catch**.

Тонкости обработки исключительных ситуаций

Обработка исключительных ситуаций в языке C++ обладает дополнительными свойствами и нюансами, которые облегчают ее применение. Эти особенности описываются ниже.

Перехват всех исключительных ситуаций

В некоторых случаях нет смысла обрабатывать отдельные типы исключительных ситуаций, а необходимо перехватывать их все подряд. Для этого достаточно применить следующий вид оператора **catch**.

```
catch(...)
{
  // Обработка всех исключительных ситуаций
}
```

Эллипсис означает, что данный оператор перехватывает все исключительные ситуации. Его применение иллюстрируется следующей программой.

```
// В этом примере перехватываются все исключительные ситуации.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try {
    if(test==0) throw test;   // Генерирует объект типа int
    if(test==1) throw 'a';    // Генерирует объект типа char
    if(test==2) throw 123.23; // Генерирует объект типа double
  } catch(...) { // Перехват всех исключительных ситуаций
    cout << "Перехват!\n";
  }
}

int main()
{
  cout << "Начало\n";
```

```
  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "Конец";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Начало
Перехват!
Перехват!
Перехват!
Конец
```

Как видим, все три исключительные ситуации, сгенерированные операторами **throw**, перехватываются одним оператором **catch**.

Оператор **catch(...)** особенно полезен, если поставить его в конце последовательности операторов **catch**. В таком случае, даже если возникнет исключительная ситуация, не предусмотренная программистом, она будет перехвачена последним оператором **catch**. Рассмотрим слегка измененный вариант предыдущей программы, в котором целочисленные исключительные ситуации перехватываются явно, а остальные ошибки отслеживаются оператором **catch(...)**.

```
// В этом примере по умолчанию используется оператор catch(...)
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try {
    if(test==0) throw test; // Генерируется объект типа int
    if(test==1) throw 'a'; // Генерируется объект типа char
    if(test==2) throw 123.23; // Генерируется объект типа double
  }
  catch(int i) { // Перехват исключительной ситуации типа int
    cout << "Перехват целого числа\n";
  }
  catch(...){ // Перехват всех остальных исключительных ситуаций
    cout << "Перехват!\n";
  }
}

int main()
{
  cout << "Начало\n";

  Xhandler(0);
  Xhandler(1);
  Xhandler(2);

  cout << "Конец";

  return 0;
}
```

Результаты работы этой программы представлены ниже.

```
Начало
Перехват целого числа
Перехват!
Перехват!
перехват!
Конец
```

Этот пример демонстрирует, что применение оператора **catch(...)** — хороший способ перехватывать все исключительные ситуации, которые нежелательно обрабатывать явно. Кроме того, оператор **catch(...)** предотвращает аварийное завершение программы при обнаружении необработанной исключительной ситуации.

Ограничения исключительных ситуаций

Программист может ограничить типы исключительных ситуаций, которые может генерировать функция в других местах программы. Фактически можно вообще запретить функции генерировать какие бы то ни было исключительные ситуации. Для этого следует добавить в определение функции раздел **throw**.

```
тип_возвращаемого_значения имя_функции(список_аргументов) throw(список_типов)
{
  // ...
}
```

Это определение функции означает, что она может генерировать лишь исключительные ситуации, перечисленные в *списке типов*. Попытка сгенерировать исключительную ситуацию любого другого типа приводит к аварийному завершению программы. Если необходимо запретить функции вообще генерировать *любые* исключительные ситуации, список типов следует оставить пустым.

Попытка сгенерировать исключительную ситуацию, не поддерживаемую функцией, сопровождается вызовом стандартной функции **unexpected()**. Затем по умолчанию вызывается функция **abort()**, и программа завершается аварийно. Однако для непредвиденной исключительной ситуации можно предусмотреть собственный обработчик.

Следующая программа демонстрирует ограничения на типы исключительных ситуаций, которые могут генерироваться функцией.

```
// Ограничение типов исключительных ситуаций,
// генерируемых функцией.
#include <iostream>
using namespace std;

// Данная функция может генерировать лишь исключительные
// ситуации, имеющие типы int, char и double.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // Генерируется объект типа int
  if(test==1) throw 'a'; // Генерируется объект типа char
  if(test==2) throw 123.23; // Генерируется объект типа double
}

int main()
{
  cout << "Начало\n";

  try {
```

```
    Xhandler(0); // Попробуйте также передать функции
                 // Xhandler() числа 1 и 2
  }
  catch(int i) {
    cout << "Перехват целого числа\n";
  }
  catch(char c) {
    cout << "Перехват строки\n";
  }
  catch(double d) {
    cout << "Перехват действительного числа\n";
  }

  cout << "Конец";

  return 0;
}
```

В этой программе функция **Xhandler()** может генерировать лишь исключительные ситуации, имеющие типы **int**, **char** и **double**. При попытке сгенерировать исключительную ситуацию любого другого типа программа завершается аварийно. (Иначе говоря, вызывается функция **unexpected()**.) Чтобы увидеть, как это происходит, удалите из списка типов спецификатор **int** и повторите компиляцию программы.

Важно четко понимать, что ограничения распространяются лишь на типы исключительных ситуаций, которые функция может генерировать в блоке **try**. Иначе говоря, блоки **try**, находящиеся *внутри* функции, могут генерировать любые исключительные ситуации. Таким образом, ограничения распространяются лишь на исключительные ситуации, генерируемые вне функции.

В следующем примере функция **Xhandler()** не может генерировать никаких исключительных ситуаций.

```
// Данная функция не может генерировать никаких
// исключительных ситуаций!
void Xhandler(int test) throw()
{
  /* Следующие операторы больше не выполняются.
     Они приводят к аварийному завершению программы. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

Повторное генерирование исключительной ситуации

Если возникает необходимость повторно возбудить исключительную ситуацию внутри ее обработчика, можно выполнить оператор **throw**, не указывая тип исключительной ситуации. В этом случае оператору **try/catch** передается текущая исключительная ситуация. Таким образом для одной и той же исключительной ситуации можно предусмотреть несколько обработчиков. Допустим, один обработчик исключительной ситуации управляет одним аспектом, а второй — остальными. Внутри блока **catch** (или функции, вызываемой из этого блока) можно лишь повторно сгенерировать исключительную ситуацию. В этом случае она передается не тому же самому оператору **catch**, а следующему. Повторное генерирование исключительной ситуации иллюстрируется следующей программой.

```
// Пример повторного генерирования исключительной ситуации.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "Привет"; // Генерируется объект типа char *
  } catch(const char *) { // Перехват объекта типа char *
    cout << "Перехват объекта типа char * внутри функции  Xhandler\n";
    throw ; // Повторное генерирование объекта типа char *
            // вне функции.
  }
}

int main()
{
  cout << "начало\n";

  try {
    Xhandler();
  } catch(const char *) {
    cout << "Перехват объекта типа char * в функции main\n";
  }

  cout << "Конец";

  return 0;
}
```

Результаты работы этой программы приводятся ниже.

```
Начало
Перехват объекта типа char * внутри функции Xhandler()
Перехват объекта типа char * внутри функции main()
Конец
```

Функции terminate() и unexpected()

Как указывалось ранее, функции **terminate()** и **unexpected()** вызываются в крайних случаях, т.е. когда обработка исключительной ситуации выполняется неверно. Эти функции принадлежат стандартной библиотеке языка C++. Их прототипы показаны ниже.

```
void terminate()
void unexpected()
```

Для вызова этих функций необходим заголовок **<exception>**.

Функция **terminate()** вызывается, если подсистема обработки исключительных ситуаций не может обнаружить подходящий оператор **catch**. Кроме того, она вызывается, если программа пытается повторно сгенерировать исключительную ситуацию, которая ранее никогда не генерировалась. Функция **terminate()** вызывается также во многих других, более запутанных ситуациях. Например, деструктор уничтожаемого объекта генерирует исключительную ситуацию в процессе раскручивания стека, который выполняется при генерировании другой исключительной ситуации. Как правило, функция **terminate()** является последним средством обработки исключительной си-

туации, если никакой другой обработчик не подходит. По умолчанию функция **terminate()** вызывает функцию **abort()**.

Функция **unexpected()** вызывается при попытке генерировать исключительную ситуацию, не указанную в разделе **throw**. По умолчанию функция **unexpected()** вызывает функцию **terminate()**.

Обработчики, связанные с функциями terminate() и unexpected()

Функции **terminate()** и **unexpected()** просто вызывают другие функции, которые на самом деле обрабатывают ошибку. Как показано выше, по умолчанию функция **terminate()** вызывает функцию **abort()**, а функция **unexpected()** — функцию **terminate()**. Таким образом, по умолчанию при возникновении исключительной ситуации обе функции прерывают выполнение программы. Однако функции **terminate()** и **unexpected()** могут вызывать другие функции. Это позволяет программе полностью контролировать подсистему обработки исключительных ситуаций.

Чтобы изменить обработчик, связанный с функцией **terminate()**, следует вызвать функцию **set_terminate()**, как показано ниже.

```
terminate_handler set_terminate(terminate_handler newhandler) throw();
```

Здесь параметр *newhandler* является указателем на новый обработчик, связанный с функцией **terminate()**. Эта функция возвращает указатель на старый обработчик, связанный с функцией **terminate()**. Новый обработчик должен иметь тип **terminate_handler**, определенный с помощью следующего оператора:

```
typedef void (*terminate_handler)();
```

Единственное предназначение обработчика terminate_handler — прекратить выполнение программы. Он не должен возвращать управление или возобновлять выполнение программы.

Чтобы изменить обработчик, связанный с функцией **unexpected()**, следует вызвать функцию **set_unexpected()**, как показано ниже.

```
unexpected_handler set_unexpected(unexpected_handler newhandler)
throw();
```

Здесь параметр *newhandler* является указателем на новый обработчик, связанный с функцией **unexpected()**. Эта функция возвращает указатель на старый обработчик, связанный с функцией **unexpected()**. Новый обработчик должен иметь тип **unexpected_handler**, определенный с помощью следующего оператора:

```
typedef void (*unexpected_handler)();
```

Этот обработчик может сам генерировать исключительные ситуации, прекращать выполнение программы или вызывать функцию **terminate()**. Однако он не должен возвращать управление программе.

Для вызова функций **set_terminate()** и **set_unexpected()** необходим заголовок **<expection>**.

Рассмотрим пример, в котором определяется обработчик, связанный с функцией **terminate()**.

```
// Определение нового обработчика,
// связанного с функцией terminate.
#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;
```

```
void my_Thandler()
{
  cout << "Внутри нового обрабочика\n";
  abort();
}

int main()
{
  // Определение нового обработчика,
  // связанного с функцией terminate.
  set_terminate(my_Thandler);

  try {
    cout << "Внутри блока try\n";
    throw 100; // Генерирование ошибки.
  } catch (double i) {
    // Не перехватывает исключительные ситуации типа int.
    // ...
  }

  return 0;
}
```

Результаты этой программы приведены ниже.

```
Внутри блока try
Внутри нового обработчика
abnormal program termination
```

Функция uncaught_exception()

Подсистема обработки исключительных ситуаций в языке C++ содержит еще одну функцию — **uncaught_exception()**. Ее прототип показан ниже.

```
bool uncaught_exception();
```

Эта функция возвращает значение **true**, если исключительная ситуация была сгенерирована, но еще не перехвачена. Если исключительная ситуация была перехвачена, функция возвращает значение **false**.

Классы exception и bad_exception

Функции из стандартной библиотеки языка C++ могут генерировать исключительные ситуации, производные от класса **exception**. Кроме того, обработчик, связанный с функцией **unexpected()**, может генерировать исключительную ситуацию **bad_exception**. Для использования этих классов требуется заголовок **<exception>**.

Применение обработки исключительных ситуаций

Система обработки исключительных ситуаций позволяет реагировать на необычные события, возникающие в ходе выполнения программы. Следовательно, обработчики исключительных ситуаций должны выполнять некие разумные действия, позволяющие ис-

править ошибку или смягчить ее последствия. Рассмотрим в качестве примера следующую простую программу. Она вводит два числа и делит первое из них на второе. Для предотвращения деления на нуль применяется обработка исключительной ситуации.

```
#include <iostream>
using namespace std;

void divide(double a, double b);

int main()
{
  double i, j;

  do {
    cout << "Введите числитель (0 означает выход): ";
    cin >> i;
    cout << "Введите знаменатель: ";
    cin >> j;
    divide(i, j);
  } while(i != 0);

  return 0;
}

void divide(double a, double b)
{
  try {
    if(!b) throw b; // Проверка деления на нуль
    cout << "Результат: " << a/b << endl;
  } catch (double b) {
    cout << "Делить на нуль нельзя.\n";
  }
}
```

Этот очень простой пример иллюстрирует принцип обработки исключительных ситуаций. Если знаменатель равен нулю, возникает исключительная ситуация. Ее обработчик не предусматривает деления (это привело бы к аварийному завершению работы программы), а просто сообщает пользователю о возникшей ошибке. Таким образом, деления на нуль можно избежать, продолжив выполнение программы. Эта схема работает и в более сложных случаях.

Обработка исключительной ситуации особенно полезна при выходе из глубоко вложенных процедур, в которых возникла катастрофическая ошибка. Язык C++ предоставляет более совершенные средства для решения этой проблемы, чем язык C, в котором применялись функции **`setjmp()`** и **`longjmp()`**.

Помните, что обработка исключительных ситуаций предназначена для обеспечения разумной реакции на возникающие ошибки. Следовательно, ситуация должна быть простой и очевидной, насколько это возможно.

Полный справочник по C++

Глава 20

Основы системы ввода-вывода

Язык C++ поддерживает две полноценные системы ввода-вывода: одна из них унаследована от языка C, а другая является объектно-ориентированной и определена в языке C++ (с этого момента мы будем называть ее просто *системой ввода-вывода языка C++*). Первая система ввода-вывода рассмотрена в части I. Теперь настало время изучить систему ввода-вывода языка C++. Системы ввода-вывода языка C и C++ являются абсолютно совместимыми. Разные компоненты каждой из систем, например, средства для работы с консолью или файлами, просто по-разному реализуют один и тот же механизм. В главе описываются основы системы ввода-вывода языка C++. Несмотря на то что в большинстве случаев рассматриваются средства работы с консолью, эту информацию можно использовать для работы с другими устройствами, например, с файлами (см. главу 21).

Поскольку система ввода-вывода, унаследованная от языка C, очень разнообразна и эффективна, возникает вопрос: "Зачем понадобилось создавать еще одну систему ввода-вывода?". Ответ прост: система ввода-вывода языка C ничего не знает об объектах. Следовательно, чтобы система ввода-вывода полностью соответствовала принципам объектно-ориентированного программирования, необходимы средства для работы с объектами, определенными пользователем. Кроме этого, система ввода-вывода языка C++ обладает еще рядом преимуществ, даже если программа не очень широко использует объекты. Честно говоря, все новые программы должны использовать только систему ввода-вывода языка C++. Старая система, унаследованная от языка C, применяется только для поддержки совместимости программ.

Сравнение старой и новой систем ввода-вывода

В настоящее время используются две библиотеки объектно-ориентированного вывода: старая, основанная на исходных спецификациях языка C++, и современная, определенная стандартом языка C++. Старая библиотека ввода-вывода поддерживается с помощью заголовка **<iostream.h>**. Новая система ввода-вывода обеспечивается заголовком **<iostream>**. Во многих отношениях эти системы ничем не отличаются, поскольку современная библиотека просто является модифицированной и улучшенной версией старой библиотеки. Основные различия между ними скрыты от наблюдателей, поскольку относятся к механизмам реализации, а не к способам применения средств ввода-вывода.

С точки зрения программиста, между старой и новой системами ввода-вывода языка C++ есть два различия. Во-первых, новая библиотека обладает дополнительными свойствами и определяет несколько новых типов данных. Следовательно, старая библиотека является подмножеством новой. Практически все программы, написанные с использованием старой библиотеки ввода-вывода, без проблем компилируются новыми компиляторами. Во-вторых, старая библиотека ввода-вывода пребывала в глобальном пространстве имен. (Напомним, что пространство имен **std** используется всеми стандартными библиотеками языка C++.) Поскольку эта библиотека морально устарела, в книге используется только новая система ввода-вывода, хотя все программы можно скомпилировать и со старой библиотекой.

Потоки

Система ввода-вывода языка C++, как и ее аналог в языке C, оперирует потоками. Это понятие подробно изучалось в главе 9, поэтому не будем повторяться. Просто подведем итоги: *поток* (stream) — это логическое устройство, получающее или передающее информацию. Поток связан с физическим устройством ввода-вывода. Все потоки функционируют одинаково, хотя физические устройства, с которыми они связа-

ны, могут быть самыми разными. Поскольку все потоки одинаковы, одна функция ввода-вывода может работать с разными типами физических устройств. Например, с помощью одной функции можно выводить данные как на принтер, так и на экран.

Классы потоков в языке C++

Как известно, для системы ввода-вывода необходим заголовок **<iostream>**. В этом заголовке определена довольно сложная иерархия классов, поддерживающих операции ввода-вывода. Сначала определяются шаблонные классы ввода-вывода. Как указывалось в главе 18, шаблонный класс представляет собой схему, в которой не уточняется тип данных, которыми она оперирует. После определения шаблонных классов можно создавать их конкретизации. Стандарт языка C++ создает две специализации шаблонных классов ввода-вывода: одну для восьмибитовых символов, а другую — для расширенных. В книге используются только классы для восьмибитовых символов, поскольку они применяются более широко. Однако описанная технология применима и к расширенным символам.

Система ввода-вывода языка C++ построена на основе двух родственных, но разных иерархий шаблонных классов. В основе первой иерархии лежит класс **basic_streambuf**, предназначенный для низкоуровневого ввода-вывода. Если в программе не используются особые процедуры ввода-вывода, класс **basic_streambuf**, как правило, не нужен. В обычных приложениях чаще всего применяется иерархия классов, построенная на основе класса **basic_ios**, обеспечивающего высокоуровневые операции ввода-вывода, проверку ошибок и анализ информации о статусе потоков. (Класс **basic_ios** является производным от класса **ios_base**, определяющего некоторые нешаблонные свойства, используемые классом **basic_ios**.) Класс **basic_ios** используется в качестве базового для нескольких производных классов, в частности **basic_istream**, **basic_ostream** и **basic_iostream**. Эти классы применяются для создания потоков, обеспечивающих соответственно ввод, вывод, а также ввод-вывод.

Как известно, библиотека ввода-вывода создает две специализации шаблонных классов, входящих в иерархию: одну для восьмибитовых символов, а другую — для расширенных. Ниже приводится список имен шаблонных классов, предназначенных для ввода-вывода обычных и расширенных символов.

Шаблонные классы	*Классы для обычных символов*	*Классы для расширенных символов*
basic_streambuf	**streambuf**	**wstreambuf**
basic_ios	**ios**	**wios**
basic_istream	**istream**	**wistream**
basic_ostream	**ostream**	**wostream**
basic_iostream	**iostream**	**wiostream**
basic_fstream	**fstream**	**wfstream**
basic_ifstream	**ifstream**	**wifstream**
basic_ofstream	**ofstream**	**wofstream**

В книге используются лишь классы, предназначенные для ввода-вывода обычных символов, поскольку именно такие операции выполняются чаще всего. Их имена совпадают с именами соответствующих классов из старой библиотеки. Именно поэтому старая и новая библиотеки ввода-вывода совместимы на уровне исходного кода.

И последнее: класс **ios** содержит большое количество функций-членов и переменных, управляющих основными операциями над потоками или наблюдающих за их выполнением. Мы будем часто на него ссылаться. Просто следует помнить: включая в программу заголовок **<iostream>**, вы получаете доступ к этому важному классу.

Встроенные потоки в языке C++

В начале выполнения программы на языке C++ автоматически открываются четыре потока.

Потоки	*Значение*	*Устройство по умолчанию*
`cin`	Стандартный ввод	Клавиатура
`cout`	Стандартный вывод	Экран
`cerr`	Стандартный вывод ошибок	Экран
`clog`	Буферизованный вариант потока `cerr`	Экран

Потоки **cin**, **cout** и **cerr** соответствуют потокам **stdin**, **stdout** и **stderr**.

По умолчанию стандартные потоки используются для взаимодействия с консолью. Однако в операционных системах, поддерживающих перенаправление потоков ввода-вывода (таких как DOS, Unix, OS/2 и Windows), стандартные потоки можно связать с другими устройствами или файлами. Для простоты в примерах мы не будем применять перенаправление потоков.

Кроме того, язык C++ определяет четыре дополнительных потока: **win**, **wout**, **werr** и **wlog**. Это версии потоков для ввода-вывода расширенных символов. Для представления расширенных символов используется тип **wchar_t** и 16-битовые значения. Как правило, расширенные символы применяются для поддержки некоторых естественных языков.

Форматированный ввод-вывод

Язык C++ позволяет выполнять операции форматированного ввода-вывода. Например, можно задать ширину поля, указать основу счисления или определить количество цифр после десятичной точки. Для форматирования данных можно применять два похожих, но разных способа. Во-первых, можно прямо обратиться к членам класса **ios**. В частности, можно самостоятельно задавать различные флаги форматирования, определенные внутри класса **ios**, или вызывать разнообразные функции-члены. Во-вторых, в выражениях ввода-вывода можно использовать специальные функции, называемые *манипуляторами* (manipulators).

Рассмотрим сначала средства форматированного ввода-вывода с помощью флагов и функций — членов класса **ios**.

Форматирование с помощью членов класса ios

Каждый поток связан с набором флагов формата, управляющих представлением информации. Класс **ios** объявляет битовую маску под названием **fmtflags**, в которой определяются следующие значения. (С технической точки зрения эти значения определены в классе **ios_base**, который является базовым по отношению к классу **ios**.)

```
adjustfield     basefield     boolalpha     dec
fixed           floatfield    hex           internal
left            oct           right         scientific
showbase        showpoint     showpos       skipws
unitbuf         uppercase
```

Эти значения используются для установки и сброса флагов формата. При работе со старыми компиляторами невозможно определить перечисление **fmtflags**. В таком случае флаги формата кодируются с помощью значений, имеющих тип **long**.

Если установлен флаг **skipws**, при вводе данных из потока разделители (пробелы, знаки табуляции и символы перехода на новую строку) игнорируются. Если этот флаг сброшен, разделители учитываются.

Если установлен флаг **left**, строки вывода выравниваются по левому краю. Если установлен флаг **right**, строки вывода выравниваются по правому краю. Если установлен флаг **internal**, между знаком числа и его первой цифрой пробелы вставляются так, чтобы число заполнило собой все поле вывода. Если ни один из этих флагов не установлен, по умолчанию выполняется выравнивание по правому краю.

По умолчанию числовые значения выводятся в десятичном виде. Однако основание системы счисления можно изменить. Для вывода восьмеричных чисел предназначен флаг **oct**. Установка флага **hex** позволяет выводить числа в шестнадцатеричном виде. Вывод чисел в десятичном формате обеспечивается флагом **dec**.

Установка флага **showbase** позволяет вывести на экран основание системы счисления. Например, при выводе шестнадцатеричных чисел значение 1F будет представлено как 0x1F.

При выводе чисел в научном формате буква **e** по умолчанию выводится как строчная. Кроме того, буква **x** в восьмеричном представлении чисел также считается строчной. Если необходимо вывести эти буквы как прописные, следует установить флаг **uppercase**.

Установка флага **showpos** позволяет вывести знак перед положительными числами.

Установка флага **showpoint** позволяет выводить десятичную точку и незначащие нули при отображении десятичных чисел.

Если установлен флаг **scintiefic**, число выводится в научном формате. Если установлен флаг **fixed**, десятичное число выводится в обычном виде. Если ни один из этих флагов не установлен, компилятор сам выбирает подходящее представление чисел.

Если установлен флаг **unitbuf**, то после каждой операции вставки буфер очищается.

Флаг **boolalpha** позволяет вводить и выводить булевские значения **true** и **false**.

Поскольку числа обычно выводятся в десятичном, восьмеричном и шестнадцатеричном виде, поля **dec**, **oct** и **hex** называют общим именем **basefield**. Аналогично поля **left**, **right** и **internal** называют **adjustfield**. Кроме того, поля **scientific** и **fixed** объединяют общим именем **floatfield**.

Установка флагов формата

Для установки флага используется функция **setf()**. Эта функция является членом класса **ios**. Она имеет следующий вид.

```
fmtflags setf(fmtflags флаги)
```

Данная функция возвращает текущее состояние флагов формата, отмеченных параметром *флаг*, и устанавливает их. Например, чтобы установить флаг **showpos**, можно применить следующий оператор.

```
stream.setf(ios::showpos);
```

Здесь имя *stream* означает поток, на который вы хотите повлиять. Обратите внимание на префикс **ios::** перед флагом **showpos**. Он необходим, поскольку флаг **showpos** является перечислимой константой, определенной в классе **ios**.

Следующая программа выводит на экран число 100, устанавливая флаги **showpos** и **showpoint**.

```
#include <iostream>
using namespace std;
```

```
int main()
{
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);

  cout << 100.0; // Выводим число +100.0

  return 0;
}
```

Следует помнить, что функция **setf()** является членом класса **ios** и влияет на потоки, созданные этим классом. Следовательно, любой вызов функции **setf()** связан с конкретным потоком. Сама по себе функция **setf()** никогда не вызывается. Иначе говоря, в языке C++ нет концепции глобального статуса формата. Каждый поток поддерживает свой собственный статус.

Хотя предыдущая программа является синтаксически правильной, ее можно переписать и сделать более эффективной. Вместо нескольких вызовов функции **setf()** можно применить к ее аргументам логическую операцию "ИЛИ". Например, предыдущие вызовы можно заменить одним.

```
// К флагам можно применять логическую операцию "ИЛИ"
cout.setf(ios::showpoint | ios::showpos);
```

Внимание! *Поскольку флаги формата определены в классе* ***ios****, доступ к ним осуществляется с помощью имени* ***ios*** *и оператора разрешения области видимости. Например, сам по себе флаг* ***showbase*** *не распознается. Необходимо уточнить его имя* ***ios::specify****.*

Сброс флагов формата

Антиподом функции **setf()** является функция **unsetf()**. Эта функция — член класса **ios** — используется для сброса одного или нескольких флагов формата и имеет следующий вид.

```
void unsetf(fmtflags флаги)
```

Данная функция сбрасывает флаги, заданные своим параметром. (Все остальные флаги сохраняют свое прежнее состояние.)

Функция **unsetf()** иллюстрируется следующей программой. Сначала она устанавливает флаги **uppercase** и **scientific**, затем выводит число 100.12 в научном формате. В данном случае научный формат числа содержит прописную букву "E". После этого программа сбрасывает флаг **uppercase** и снова выводит число 100.12 в научном формате, на этот раз используя строчную букву "e".

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::uppercase | ios::scientific);

  cout << 100.12;  // Выводим число 1.0012E+02

  cout.unsetf(ios::uppercase); // Сбрасываем флаг uppercase

  cout << " \n" << 100.12; // Выводим число 1.0012e+02

  return 0;
}
```

Перегруженная форма функции setf()

Функция **setf()** имеет перегруженную форму.

```
fmtflags setf(fmtflags флаги1, fmtflags флаги2);
```

В этой версии изменяются только флаги, заданные параметром *флаги2*. Сначала они сбрасываются, а затем устанавливаются в соответствии со статусами флагов, заданных параметром *флаги1*. Обратите внимание на то, что даже если параметры *флаги1* и *флаги2* относятся к разным флагам, изменяются только флаги, заданные параметром *флаги2*. Функция возвращает предыдущее состояние флагов.

```
#include <iostream>
using namespace std;

int main( )
{
  cout.setf(ios::showpoint | ios::showpos, ios::showpoint);

  cout << 100.0; // Выводит на экран число 100.0, а не +100.0

  return 0;
}
```

Программа устанавливает флаг **showpoint**, а не **showpos**, поскольку флаг **showpos** не указан во втором параметре.

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::hex, ios::basefield);

  cout << 100; // Выводит на экран число 64

  return 0;
}
```

В этой программе сначала сбрасываются флаги **basefield** (т.е. флаги **dec**, **oct** и **hex**), а затем устанавливается флаг **hex**.

Помните, что состояние флагов, заданных параметром *флаги1*, влияет только на флаги, заданные параметром *флаги2*. Например, в следующей программе первая попытка установить флаг *showpos* оказывается неудачной.

```
// Эта программа не работает.
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos, ios::hex); // Ошибка, флаг
                                     // showpos не установлен.

  cout << 100 << '\n'; // Выводит на экран число 100, а не +100

  cout.setf(ios::showpos, ios::showpos); // Правильно

  cout << 100; // Теперь на экран выводится число +100.
```

```
  return 0;
}
```

Учтите, что в большинстве случаев для сброса флагов используется функция **unsetf()**, а для установки флага вызывается функция **setf()** с одним параметром. В более сложных случаях, например, при выводе основания системы счисления следует вызывать функцию **setf(fmtflags, fmtflags)**. Кроме того, иногда программисты создают шаблон всех флагов формата, описывающий представление чисел, а затем пытаются изменить один или два флага. В этом случае шаблон задают с помощью параметра *флаги1*, а изменяемые флаги перечисляют в параметре *флаги2*.

Проверка флагов форматирования

Иногда необходимо точно определить текущее состояние формата, не изменяя его флаги. Для этого в классе **ios** используется функция-член **flags()**, которая возвращает текущее состояние каждого флага формата. Ее прототип имеет следующий вид.

```
fmtflags flags();
```

В следующей программе эта функция применяется для установки флагов форматирования, связанных с потоком **cout**. Обратите особое внимание на функцию **showflags()**. Она может оказаться полезной.

```
#include <iostream>
using namespace std;

void showflags() ;

int main()
{
  // Показать состояние флагов формата,
  // предусмотренное по умолчанию.
  showflags();

  cout.setf(ios::right | ios::showpoint | ios::fixed);

  showflags();

  return 0;
}

// Функция выводит на экран состояние флагов формата.
void showflags()
{
  ios::fmtflags f;
  long i;

  f = (long) cout.flags(); // Получить состояние флагов.

  // Проверить каждый флаг
  for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";

  cout << " \n";
}
```

Ниже показаны результаты работы этой программы. (Эти установки зависят от компилятора.)

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 1 0 0 0 1 0 1 0 0 1 0 0 0 1
```

Установка всех флагов

Функция **flags()** имеет вторую форму, позволяющую устанавливать все флаги формата, связанные с потоком. Прототип этой версии функции **flags()** показан ниже.

```
fmtflags flags(fmtflags f);
```

В этой версии параметр *f* задает битовый шаблон флагов формата, связанных с конкретным потоком. Таким образом, изменяются сразу все флаги форматирования. Функция возвращает предыдущее состояние флагов.

Эта версия функции **flags()** иллюстрируется следующей программой. Сначала она создает битовую маску, в которой установлены флаги **showpos**, **showbase**, **oct** и **right**. Все остальные флаги сброшены. Затем для установки флагов форматирования, связанных с потоком **cout**, используется функция **flags()**. Функция **showflags()** выводит на экран состояние флагов форматирования. (Это та же функция, которая была определена в предыдущем примере.)

```
#include <iostream>
using namespace std;

void showflags();

int main()
{
  // Выводим на экран состояние флагов формата,
  // предусмотренное по умолчанию.
  showflags();

  // Флаги showpos, showbase, oct и right установлены,
  // остальные сброшены.
  long f = ios::showpos | ios::showbase | ios::oct | ios::right;
  cout.flags(f);  // Задаем состояние всех флагов.

  showflags();

  return 0;
}
```

Применение функций width(), precision() и fill()

В классе **ios** предусмотрены три функции-члена, позволяющие изменять ширину поля вывода, точность и символ-заполнитель. Они называются **width()**, **precision()** и **fill()** соответственно. Рассмотрим каждую из них в отдельности.

По умолчанию количество позиций, которые занимает число при выводе, равно количеству символов, из которых оно состоит. Однако существует возможность изменить минимальную ширину поля вывода, используя функцию **width()**. Ее прототип показан ниже.

```
streamsize width(streamsize w);
```

Функция возвращает предыдущую ширину поля, а ее новое значение задается параметром *w*. В некоторых реализациях ширина поля должна задаваться при каждом вы-

воде. Если этого не сделать, используется ее значение, предусмотренное по умолчанию. Тип **streamsize** представляет собой один из вариантов типа **int**, причем конкретный выбор зависит от компилятора.

Если число не целиком заполняет поле, оно дополняется символом-заполнителем (по умолчанию — пробелом), чтобы длина числа совпадала с шириной поля. Если число не помещается в поле вывода, оно все равно выводится полностью. Усечение чисел не производится.

При выводе действительных чисел можно задавать количество цифр после десятичной точки (точность числа), используя функцию **precision()**. Ее прототип представлен ниже.

```
streamsize precision(streamsize p);
```

Функция возвращает старое значение, а новое количество цифр после десятичной точки задается параметром *p*. По умолчанию после десятичной точки выводится 6 цифр. В некоторых реализациях это значение следует задавать перед каждым выводом, иначе будет использоваться точность, предусмотренная по умолчанию.

Кроме того, если поле вывода не заполнено, оно автоматически дополняется пробелами. Символ-заполнитель можно изменить, использовав функцию **fill()**. Ее прототип имеет следующий вид.

```
char fill(char ch);
```

Функция **fill()** возвращает старый символ-заполнитель, а новым заполнителем становится символ *ch*.

Рассмотрим программу, иллюстрирующую применение этих функций.

```
#include <iostream>
using namespace std;

int main()
{
  cout.precision(4) ;
  cout.width(10);

  cout << 10.12345 << "\n";  // Выводит на экран 10.12

  cout.fill('*');

  cout.width(10);
  cout << 10.12345 << "\n"; // Выводит на экран *****10.12

  // Ширина поля вывода распространяется и на строки
  cout.width(10);
  cout << "Hi!" << "\n"; // Выводит *******Hi!
  cout.width(10);
  cout.setf(ios::left); // Выравнивает по левому краю
  cout << 10.12345; // Выводит на экран 10.12*****

  return 0;
}
```

Результат работы этой программы выглядит так.

```
     10.12
*****10.12
*******Hi!
10.12*****
```

Функции **width()**, **precision()** и **fill()** имеют перегруженные версии, получающие, но не изменяющие соответствующие параметры потока.

```
char fill();
streamsize width();
streamsize precision();
```

Применение манипуляторов формата

Другой способ изменения формата основан на применении специальных функций, называемых *манипуляторами* (manipulators). Их можно включать в операторы ввода-вывода. Стандартные манипуляторы перечислены в табл. 20.1.

Таблица 20.1. Манипуляторы

Манипуляторы	*Предназначение*	*Ввод-вывод*
boolalpha	Устанавливает флаг **boolalpha**	Ввод-вывод
dec	Устанавливает флаг **dec**	Ввод-вывод
endl	Выводит символ перехода на новую строку и очищает буфер	Вывод
ends	Выводит нулевой байт	Вывод
fixed	Устанавливает флаг **fixed**	Вывод
flush	Очищает буфер	Вывод
hex	Устанавливает флаг **hex**	Ввод-вывод
internal	Устанавливает флаг **internal**	Вывод
left	Устанавливает флаг **left**	Вывод
noboolalpha	Сбрасывает флаг **boolalpha**	Ввод-вывод
noshowbase	Сбрасывает флаг **showbase**	Вывод
noshowpoint	Сбрасывает флаг **showpoint**	Вывод
noshowpos	Сбрасывает флаг **showpos**	Вывод
noskipws	Сбрасывает флаг **skipws**	Ввод
nounitbuf	Сбрасывает флаг **unitbuf**	Вывод
nouppercase	Сбрасывает флаг **uppercase**	Вывод
oct	Устанавливает флаг **oct**	Ввод-вывод
resetiosflags(fmtflags *f*)	Сбрасывает флаги, указанные параметром *f*	Ввод-вывод
right	Устанавливает флаг **right**	Вывод
scientific	Устанавливает флаг **scientific**	Вывод
setbase(int *base*)	Задает основание системы счисления, указанное параметром *base*	Ввод-вывод
setfill(int *ch*)	Задает символ-заполнитель *ch*	Вывод
setiosflags(fmtflags f)	Устанавливает флаги, указанные параметром *f*	Ввод-вывод
setprecision(int *p*)	Задает количество цифр после десятичной точки	Вывод
setw(int *w*)	Задает ширину поля, указанную параметром *w*	Вывод
showbase	Устанавливает флаг **showbase**	Вывод
showpoint	Устанавливает флаг **showpoint**	Вывод
showpos	Устанавливавт флаг **showpos**	Вывод
skipws	Устанавливает флаг **skipws**	Ввод
unitbuf	Устанавливает флаг **unitbuf**	Вывод
uppercase	Устанавливавт флаг **uppercase**	Вывод
ws	Игнорирует ведущие разделители	Ввод

Для доступа к манипуляторам, получающим параметры (например, к функции **setw()**), необходимо включить в программу заголовок **<iomanip>**.

Рассмотрим пример, иллюстрирующий применение манипуляторов.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << hex << 100 << endl;

  cout << setfill('?') << setw(10) << 2343.0;

  return 0;
}
```

Результат работы программы приведен ниже.

```
64
??????2343
```

Обратите внимание на то, как манипуляторы включаются в операторы вывода Кроме того, если манипулятор не имеет аргументов (например, **endl**), скобки после него не ставятся, поскольку его имя является адресом, передаваемым перегруженному оператору "<<".

Для сравнения приведем функционально эквивалентную версию предыдущей программы, в которой используются функции — члены класса **ios**.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout.setf(ios::hex, ios::basefield);
  cout << 100 << "\n";  // Число 100 в шестнадцатеричном виде

  cout.fill('?');
  cout.width(10);
  cout << 2343.0;

  return 0;
}
```

Как видим, основное преимущество манипуляторов над функциями — членами класса **ios** заключается в компактности кода.

Используя манипулятор **setiosflags()**, можно установить сразу все флаги формата, связанные с потоком. Например, в следующей программе манипулятор **setiosflags()** применяется для установки флагов **showbase** и **showpos**.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setiosflags(ios::showpos);
```

```
  cout << setiosflags(ios::showbase);
  cout << 123 << " " << hex << 123;

  return 0;
}
```

Манипулятор **setiosflags()** и функция **setf()** выполняют одинаковые операции.

Манипулятор **boolalpha** более интересен. Он позволяет вводить и выводить логические значения, используя ключевые слова **true** и **false**, а не числа.

```
#include <iostream>
using namespace std;

int main()
{
  bool b;

  b = true;
  cout << b << " " << boolalpha << b << endl;

  cout << "Введите булево значение: ";
  cin >> boolalpha >> b;
  cout << "Вы ввели следующее значение:  " << b;

  return 0;
}
```

Рассмотрим примерный сценарий ее работы.

```
1 true
Введите булево значение: false
Вы ввели следующее значение: false
```

Перегрузка операторов "<<" и ">>"

Как известно, операторы "<<" и ">>" в языке C++ перегружены и позволяют вводить и выводить данные встроенных типов. Кроме того, операторы "<<" и ">>" можно перегрузить так, чтобы они выводили объекты классов, определенных пользователем.

Оператор вывода "<<" называется *оператором вставки* (insertion operator), потому что он вставляет символы в поток. Аналогично оператор ввода ">>" называется *оператором извлечения* (extraction operator), потому что он извлекает символы из потока. Функции, перегружающие операторы вставки и извлечения, называются *функциями вставки* (inserters) и *извлечения* (extractors) соответственно.

Создание собственных функций вставки

Создание собственных функций вставки не вызывает особых трудностей. Все они имеют следующий вид.

```
ostream &operator<<(ostream &stream, класс obj)
{
  // Тело функции вставки
}
```

Обратите внимание на то, что функция возвращает ссылку на поток типа **ostream**. (Напомним, что класс **ostream** является производным от класса **ios**.) Кроме того, первый

параметр функции является ссылкой на поток вывода. Второй параметр представляет собой объект, подлежащий вставке. (Второй параметр может быть также ссылкой на этот объект.) Прежде чем закончить свою работу, функция вставки должна вернуть *указатель на поток*. Это позволяет использовать функции вставки внутри сложных выражений.

Внутри функции вставки можно выполнять любые операции. Собственно, именно они определяют сущность функции. Однако функцию вставки не следует слишком усложнять. Например, совершенно нецелесообразно заставлять функцию вставки попутно вычислять число "пи" с точностью до 30 десятичных цифр!

Продемонстрируем функцию вставки на примере объектов класса **phonebook**.

```
class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};
```

В этом классе хранятся имена и номера телефонов. Посмотрим, как выглядит функция вставки для этого класса.

```
// Вывести на экран имя и номер телефона.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // Функция должна воэвращать ссылку на поток
}
```

Ниже приведена короткая программа, иллюстрирующая применение функции для вставки объектов класса **phonebook** в поток вывода.

```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};
```

```
// Вывести на экран имя и номер телефона.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // Функция должна возвращать ссылку на поток
}

int main()
{
  phonebook a("Тед", 111, 555, 1234);
  phonebook b("Алиса", 312, 555, 5768);
  phonebook c("Том", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

Результаты работы этой программы таковы.

```
Тед (111) 555-1234
Алиса (312) 555-5768
Том (212) 555-9991
```

Обратите внимание на то, что функция вставки не является членом класса **phonebook**. На первый взгляд, это кажется странным, однако причину легко понять. Если операторная функция является членом класса, ее левым операндом (неявно передаваемым с помощью указателя **this**) является вызывающий объект. Кроме того, этот операнд является *объектом класса*, членом которого является сама операторная функция. Изменить эту ситуацию невозможно. Итак, если перегруженная операторная функция является членом класса, ее левый операнд является *потоком*, а правый — объектом данного класса. Следовательно, перегруженные функции вставки не могут быть членами класса, для которого они перегружаются. Переменные **name**, **areacode**, **prefix** и **num** в предыдущей программе являются открытыми, поэтому функция вставки имеет к ним доступ.

То, что перегруженные функции вставки не могут быть членами класса, для которого они определяются, кажется серьезным недостатком. Если функция вставки не принадлежит классу, как она может обращаться к его закрытым членам? В предыдущей программе все члены классы были открытыми. Однако инкапсуляция является одним из основных принципов объектно-ориентированного программирования, следовательно, нельзя требовать, чтобы все члены класса всегда были открытыми. К счастью, эта дилемма имеет решение: необходимо сделать функцию вставки дружественной по отношению к указанному классу. В этом случае первый аргумент перегруженной функции вставки может оставаться потоком, а сама функция получит доступ к закрытым членам класса, для которого она определена. Рассмотрим пример, иллюстрирующий этот прием.

```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
  // Теперь переменные-члены закрыты
  char name[80];
```

```
  int areacode;
  int prefix;
  int num;
public:
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
};

// Выводит на экран имя и номер телефона.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // Функция должна возвращать ссылку на поток
}

int main()
{
  phonebook a("Тед", 111, 555, 1234);
  phonebook b("Алиса", 312, 555, 5768);
  phonebook c("Том", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

Определяя тело функции вставки, старайтесь сделать его как можно более универсальным. Например, функцию вставки из предыдущего примера можно применять к любому потоку, поскольку она направляет данные в объект **stream**, являющийся потоком, вызывающим функцию вставки. Можно было бы написать операторы

```
stream << o.name << " ";
```

или

```
cout << o.name << " ";
```

Однако в этом случае поток вывода оказался бы слишком жестко заданным. Исходная версия программы работает с любым потоком, включая поток, связанный с файлом. Хотя в некоторых ситуациях, например, при выводе на специальное устройство у программиста нет выбора, и он может "зашить" поток в программу, в большинстве случаев это не так. Чем универсальнее функция вставки, она ценнее.

На заметку

Функция вставки для класса **phonebook** *работает прекрасно, пока не встретится номер наподобие 0034. В этом случае ведущие нули игнорируются и не выводятся на экран. Чтобы исправить этот недостаток, необходимо либо преобразовать переменную* **num** *в строку, либо задать в качестве символа-заполнителя цифру 0 и применить функцию* **width()** *для генерирования ведущих нулей. Читатели могут сами справиться с этой задачей.*

Прежде чем перейти к функциям извлечения, рассмотрим еще один пример функции вставки. Как известно, функции вставки позволяют вывести любые осмысленные данные. Например, такая функция для класса, являющегося частью системы автоматизированного проектирования (CAD — Computer-Aided Design), выводит инструкции плоттера. Другая функция вставки может генерировать графические изображения. Функции вставки для Windows-приложений могут выводить на экран диалоговые окна. Чтобы продемонстрировать вывод объектов, которые отличаются от простого текста, приведем следующую программу, которая рисует прямоугольники на экране. (Поскольку графические библиотеки в языке C++ не определены, программа использует символы, но их можно легко заменить графическими изображениями, если конкретная система позволяет это сделать.)

```
#include <iostream>
using namespace std;

class box {
  int x, y;
public:
  box(int i, int j) { x=i; y=j; }
  friend ostream &operator<<(ostream &stream, box o);
};

// Выводит прямоугольник.
ostream &operator<<(ostream &stream, box o)
{
  register int i, j;

  for(i=0; i<o.x; i++)
    stream << "*";

  stream << "\n";

  for(j=1; j<o.y-1; j++) {
    for(i=0; i<o.x; i++)
      if(i==0 || i==o.x-1) stream << "*";
      else stream << " ";
    stream << "\n";
  }

  for(i=0; i<o.x; i++)
     stream << "*";
  stream << "\n";

  return stream;
}

int main()
{
  box a(14, 6), b(30, 7), c(40, 5);

  cout << "Вот несколько прямоугольников:\n";
  cout << a << b << c;

  return 0;
}
```

Программа выводит на экран следующие строки.

```
Вот несколько прямоугольников:
**************
*            *
*            *
*            *
*            *
**************
*****************************
*                           *
*                           *
*                           *
*                           *
*                           *
*****************************
****************************************
*                                      *
*                                      *
*                                      *
****************************************
```

Создание собственных функций извлечения

Функции извлечения являются антиподами функций вставки. Они имеют следующий вид.

```
istream &operator>>(istream &stream, класс Eobj)
{
   // Тело функции извлечения
   return stream;
}
```

Функция извлечения возвращает ссылку на поток типа **istream**, т.е. на поток ввода. Ее первым параметром должна быть ссылка на поток типа **istream**. Обратите внимание на то, что вторым параметром должна быть ссылка на объект класса, для которого перегружена функция извлечения. Это позволяет модифицировать объект при выполнении операции ввода (извлечения).

Продолжая совершенствовать класс **phonebook**, напишем для него функцию извлечения.

```
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Введите имя: ";
  stream >> o.name;
  cout << "Введите код города: ";
  stream >> o.areacode;
  cout << "Введите префикс: ";
  stream >> o.prefix;
  cout << "Введите номер: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}
```

Хотя эта функция предназначена для ввода, при необходимости ее можно применять для вывода данных и выполнения других операций. Однако лучше руководствоваться здравым смыслом и не использовать ее по прямому назначению.

Рассмотрим пример использования функции извлечения для класса **phonebook**.

```
#include <iostream>
#include <cstring>
using namespace std;

class phonebook {
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook() { };
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Выводит на экран имя и номер телефона.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // Функция должна возвращать ссылку на поток.
}

// Вводит имя и номер телефона.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Введите имя: ";
  stream >> o.name;
  cout << "Введите код города: ";
  stream >> o.areacode;
  cout << "Введите префикс: ";
  stream >> o.prefix;
  cout << "Введите номер: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}

int main()
{
  phonebook a;

  cin >> a;

  cout << a;

  return 0;
}
```

Фактически функция извлечения для класса **phonebook** не слишком эффективна, поскольку операторы **cout** необходимы, только если поток ввода связан с интерактивным устройством, например консолью (т.е. когда потоком ввода является поток **cin**). Например, если функция извлечения применяется к потоку, связанному с файлом, операторы **cout** становятся непригодными. Забавы ради можете попробовать отменить выполнение операторов **cout**, если потоком ввода не является поток **cin**. Например, можно добавить в программу операторы вида

```
if(stream == cin) cout << "Введите имя: ";
```

Теперь приглашение ко вводу появится только в том случае, если устройством вывода является экран.

Создание собственных манипуляторов

Систему ввода-вывода можно усовершенствовать, создав свои собственные манипуляторы. Эта возможность является важной по двум причинам. Во-первых, можно сосредоточить несколько операций ввода-вывода в одном манипуляторе. Например, довольно часто в программах выполняется одна и та же последовательность операций ввода-вывода. В этом случае их можно объединить. Это позволяет упростить код и предотвратить случайные ошибки. Во-вторых, собственный манипулятор может понадобиться при работе с нестандартными устройствами ввода-вывода. Например, манипулятор можно применять для передачи команд специальному принтеру или системе оптического распознавания.

Собственные манипуляторы полностью соответствуют принципам объектно-ориентированного программирования, но они могут оказаться полезными и для программ, созданных в рамках процедурного программирования. Как мы впоследствии убедимся, собственные манипуляторы могут помочь более эффективно и просто выполнить любую программу, интенсивно выполняющую ввод и вывод данных.

Как известно, существуют два типа манипуляторов: для ввода и для вывода. Кроме того, манипуляторы разделяются на получающие аргументы и не имеющие их. Честно говоря, процедуры, необходимые для создания параметризованных манипуляторов, сильно зависят от конкретного компилятора и даже от его версии. По этой причине, прежде чем приступать к разработке параметризованного манипулятора, следует обратиться к документации, описывающей компилятор. В то же время манипуляторы, не имеющие параметров, создаются довольно просто, причем эта процедура для всех компиляторов одинакова.

Скелет манипулятора, не имеющего параметров, выглядит так.

```
ostream &имя_манипулятора(ostream &stream)
{
   // Код
   return stream;
}
```

Обратите внимание на то, что манипулятор возвращает ссылку на поток типа **ostream**. Это позволяет использовать манипулятор внутри более сложных выражений. Следует иметь в виду, что, хотя формально манипулятор имеет один аргумент, представляющий собой ссылку на поток, при вставке манипулятора в оператор вывода этот аргумент не указывается.

Рассмотрим программу, в которой описан манипулятор **sethex()**, устанавливающий флаг **showbase** и порождающий вывод чисел в шестнадцатеричном формате.

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;

// Простой манипулятор вывода.
ostream &sethex(ostream &stream)
{
  stream.setf(ios::showbase);
  stream.setf(ios::hex, ios::basefield);

  return stream;
}

int main()
{
  cout << 256 << " " << sethex << 256;

  return 0;
}
```

Эта программа выводит на экран следующие числа.

```
256 0x100
```

Очевидно, что манипулятор **sethex** используется как часть оператора вывода.

Собственные манипуляторы не должны быть слишком сложными. Например, простые манипуляторы **la()** и **ra()** выводят на экран стрелки, направленные влево и вправо.

```
#include <iostream>
#include <iomanip>
using namespace std;

// Стрелка, направленная вправо
ostream &ra(ostream &stream)
{
  stream << "-------> ";
  return stream;
}

// Стрелка, направленная влево
ostream &la(ostream &stream)
{
  stream << " <-------";
  return stream;
}

int main()
{
  cout << "Остаток на счету " << ra << 1233.23 << "\n";
  cout << "Задолженность " << ra << 567.66 << la;

  return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Остаток на счету -------> 1233.23
Задолженность -------> 567.66 <-------
```

Если бы эти стрелки приходилось часто выводить вручную, вы быстро бы устали.

Манипуляторы вывода особенно полезны при передаче команд специальным устройствам. Например, принтер может получать большое количество разнообразных команд,

изменяющих размер шрифта или его начертание, а также положение печатающей головки. Если эти команды выполняются часто, их лучше поручить манипулятору.

Все манипуляторы ввода, не имеющие параметров, выглядят следующим образом.

```
istream &имя_манипулятора(ostream &stream)
{
   // Код
   return stream;
}
```

Манипулятор ввода получает ссылку на поток, для которого он вызывается, а затем возвращает ее вызывающему модулю.

В следующей программе манипулятор **getpass()** предназначен для генерации сигнала и ввода пароля.

```
#include <iostream>
#include <cstring>
using namespace std;

// Простой манипулятор ввода.
istream &getpass(istream &stream)
{
  cout << '\a';  // Звуковой сигнал
  cout << "Введите пароль: ";

  return stream;
}

int main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "пароль"));

  cout << "Вход в систему выполнен\n";

  return 0;
}
```

Обратите внимание на то, что манипулятор должен возвращать ссылку на объект класса **istream**, иначе его будет невозможно использовать в цепочках операторов ввода-вывода.

Полный справочник по

Глава 21

Файловая система

Несмотря на то что система ввода-вывода языка C++ в целом представляет собой единый механизм, система файлового ввода-вывода имеет свои особенности. Частично это объясняется тем, что на практике чаще всего используются файлы на жестком диске, возможности которых значительно отличаются от всех других устройств. Однако следует иметь в виду, что файловый ввод-вывод является лишь частью общей системы ввода-вывода, и большая часть материала, изложенного в главе, относится к потокам, которые могут быть связаны с другими устройствами.

Заголовок <fstream> и классы файлов

Для реализации файлового ввода-вывода в программу следует включить заголовок **<fstream>**. В нем определены некоторые классы, в частности, **ifstream**, **ofstream** и **fstream**. Эти классы являются производными от классов **istream**, **ostream** и **iostream** соответственно. Следует помнить, что классы **istream**, **ostream** и **iostream**, в свою очередь, являются производными от класса **ios**, рассмотренного в предыдущей главе. Файловая система использует также класс **filebuf**, предоставляющий низкоуровневые средства управления файловым потоком. Обычно класс **filebuf** непосредственно не применяется, однако он является составной частью других классов.

Открытие и закрытие файла

В языке C++ открытие файла означает его связывание с потоком. Следовательно, сначала необходимо получить поток. Существуют три вида потоков: ввода, вывода и ввода-вывода. Для того чтобы создать поток ввода, необходимо объявить поток, представляющий собой объект класса **ifstream**. Для генерации потока вывода необходимо объявить поток, представляющий собой объект класса **ofstream**. Потоки, осуществляющие ввод и вывод, объявляются как объекты класса **fstream**. Например, в следующем фрагменте создается поток ввода, поток вывода и поток ввода-вывода.

```
ifstream in;  // Ввод
ofstream out; // Вывод
fstream io;   // Ввод и вывод
```

Созданный поток можно связать с файлом с помощью функции **open()**. Эта функция является членом каждого из трех потоковых классов. Ее прототипы в каждом классе показаны ниже.

```
ifstream::open(const char *filename, ios::openmode mode = ios::in);
ifstream::open(const char *filename, ios::openmode mode =
                                        ios::out | ios::trunc);
ifstream::open(const char *filename, ios::openmode mode =
                                        ios::in | ios::out);
```

Здесь параметр *filename* задает имя файла. Он может содержать путь к этому файлу. Значение параметра *mode* определяет способ открытия файла. Этот параметр может принимать следующие значения, описанные в функции **openmode**.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Данные значения можно комбинировать с помощью логической операции "ИЛИ".

Если задается значение **ios::app**, все результаты дописываются в конец файла, открытого для вывода. Если указано значение **ios::ate**, при открытии выполняется поиск конца файла. Несмотря на это, запись производится в любое место файла.

Если задается значение **ios::in**, файл открывается для ввода, а если **ios::out** — для вывода.

Значение **ios::binary** позволяет открыть файл в бинарном режиме. По умолчанию все файлы открываются в текстовом режиме. В этом случае производится преобразование некоторых символов, например, эскейп-последовательность "возврат каретки" и "прогон бумаги" преобразуется в символ перехода на новую строку. Однако, если файл открыт в бинарном режиме, преобразование символов не производится. Следует помнить, что любой файл можно открыть как в текстовом, так и в бинарном режиме. Единственное различие между ними заключается в том, производится преобразование символов или нет.

Значение **ios::trunc** сигнализирует, что предыдущее содержимое существующего файла с тем же именем будет уничтожено, а длина файла уменьшена до нуля. При открытии потока вывода с помощью класса **ofstream** содержимое любого существующего файла с указанным именем стирается.

В следующем фрагменте открывается текстовый файл для вывода.

```
ofstream out;
out.open("test", ios::out);
```

Однако функция **open()** редко применяется для открытия файлов, так как для каждого типа потока параметр *mode* имеет значения, заданные по умолчанию. Прототипы функции **open** демонстрируют, что значение параметра *mode*, заданное по умолчанию, в классе **ifstream** равно **ios::in**, в классе **ofstream** — **ios::out | ios::trunc**, а в классе **fstream** — **ios::in | ios::out**. По этой причине предыдущий вызов функции **open** обычно записывают так:

```
out.open("test"); // Текстовый файл для вывода
```

На заметку *В зависимости от компилятора параметр **mode** для функции **fstream::open()** может иметь значение по умолчанию, отличное от **in | out**. Следовательно, иногда его необходимо задавать явно.*

Если поток применяется в логических выражениях, в случае отказа функции **open()** ему присваивается значение **false**. Таким образом, перед использованием файла следует убедиться, что он успешно открыт. Для этого можно воспользоваться следующими операторами.

```
if(!mystream) {
  cout << "Невозможно открыть файл.\n";
  // Обработка ошибки
}
```

Однако чаще всего функцию **open()** не применяют, поскольку классы **ifstream**, **ofstream** и **fstream** содержат конструкторы, автоматически открывающие файл. Параметры этих конструкторов принимают те же значения по умолчанию, что и функция **open()**. По этой причине файлы обычно открывают следующим образом.

```
ifstream mystream("myfile"); // Открытие файла для ввода
```

Если по какой-либо причине файл открыть не удалось, связанному с ним потоку присваивается значение **false**. Следовательно, если некий конструктор вызывает функцию **open()**, следует убедиться, что файл действительно открыт, проверив значение потока.

Чтобы проверить, открыт ли файл, можно вызвать функцию **is_open()**, являющуюся членом класса **fstream**, **ifstream** и **ofstream**. Она имеет следующий прототип.

```
bool is_open();
```

Если поток связан с открытым файлом, эта функция возвращает значение **true**, в противном случае она возвращает значение **false**. Например, следующий фрагмент проверяет, открыт ли файл, связанный с потоком **mystream**.

```
if(!mystream.is_open()) {
  cout << "Файл не открыт.\n";
  // ...
```

Чтобы закрыть файл, следует вызвать функцию **close()**. Например, чтобы закрыть файл, связанный с потоком **mystream**, можно применить следующий оператор.

```
mystream.close();
```

Функция **close()** не имеет параметров и не возвращает никаких значений.

Чтение и запись текстовых файлов

Чтение и запись текстовых файлов осуществляются очень легко. Для этого достаточно применить операторы "<<" и ">>", как это обычно делается для консольного ввода-вывода, только вместо потоков **cin** и **cout** необходимо подставить поток, связанный с файлом. Например, следующая программа создает короткий файл, содержащий название предмета и его стоимость.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out("INVNTRY"); // Текстовый файл для вывода.

  if(!out) {
    cout << "Невозможно открыть файл INVENTORY.\n";
    return 1;
  }

  out << "Радиоприемники " << 39.95 << endl;
  out << "Тостеры " << 19.95 << endl;
  out << "Миксеры " << 24.80 << endl;

  out.close();
  return 0;
}
```

Программа, приведенная ниже, считывает файл, созданный предыдущей программой, и выводит его содержимое на экран.

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main()
{
  ifstream in("INVNTRY"); // Ввод

  if(!in) {
    cout << "Невозможно открыть файл INVENTORY.\n";
    return 1;
  }

  char item[20];
  float cost;

  in >> item >>  cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";

  in.close();
  return 0;
}
```

Чтение и запись файлов с помощью операторов "**<<**" и "**>>**" напоминает применение функций **fprintf()** и **fscanf()** из языка С. Формат записей в файле ничем не отличается от формата вывода на экран.

Рассмотрим еще один пример работы с файлами на жестком диске. Эта программа считывает строки, введенные с клавиатуры, и записывает их на диск. Программа останавливается, если пользователь ввел знак восклицания. При вызове программы следует указать в командной строке имя файла, предназначенного для вывода.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Вызов: output <имя файла>\n";
    return 1;
  }

  ofstream out(argv[1]); // Текстовый файл для вывода.

  if(!out) {
    cout << "Невозможно открыть файл для вывода.\n";
    return 1;
  }

  char str[80];
  cout << "Запись строк на жесткий диск.\
      Для прекращения работы введите восклицательный знак.\n";

  do {
    cout << ": ";
    cin >> str;
    out << str << endl;
  } while (*str != '!');
```

```
    out.close();
    return 0;
}
```

Считывая файлы с помощью оператора ">>", имейте в виду, что некоторые символы при вводе трансформируются. Например, разделители игнорируются. Чтобы предотвратить преобразование символов при чтении, файл следует открыть в бинарном режиме и применить функции, описанные в следующем разделе.

Если при вводе достигается конец файла, потоку, связанному с файлом, присваивается значение **false**. (Эта ситуация иллюстрируется в следующем разделе.)

Бесформатный и бинарный ввод-вывод

Итак, чтение и запись форматированных текстовых файлов не вызывает никаких затруднений, хотя это не самый эффективный способ работы с файлами. Кроме того, иногда возникает необходимость хранить бесформатные данные, а не текст. Рассмотрим функции, предназначенные для работы с такими данными.

Выполняя с файлом бинарные операции, следует убедиться, что он открыт в режиме **ios::binary**. Бесформатные данные могут храниться и в текстовом файле, но в этом случае при чтении некоторые символы будут преобразованы. Бинарные файлы применяются именно для того, чтобы этого избежать.

Сравнение символов и байтов

При изучении бесформатного ввода-вывода необходимо учитывать следующее. Многие годы ввод-вывод в языках C и C++ был *байтовым* (byte oriented). Это происходило потому, что символ (**char**) является эквивалентом байта, и потоки ввода-вывода были символьными. Однако с появлением расширенных символов (**wchar_t**) и связанных с ними потоков систему ввода-вывода языка C++ нельзя назвать байтовой. Теперь ее следует называть *символьной* (character oriented). Разумеется, потоки обычных символов (**char**) остаются байтовыми, особенно при обработке нетекстовых данных. Однако эквивалентность понятий "символ" и "байт" больше не гарантируется.

Как указывалось в главе 20, в книге используются только символьные (т.е. байтовые) потоки, поскольку они наиболее широко распространены. Они обеспечивают более простую обработку бесформатных файлов, так как символьные потоки сохраняют однозначное соответствие между символами и байтами.

Функции put() и get()

Один из способов чтения и записи бесформатных файлов основан на применении функций **put()** и **get()**. Эти функции оперируют символами. Точнее говоря, функция **get()** считывает символ, а функция **put()** — записывает его. Разумеется, если файл открыт в бинарном режиме, то при считывании символа (а не расширенного символа), эти функции считывают и записывают байты.

Функция **get()** имеет несколько форм, однако чаще всего используется ее следующая версия."Класс:ostream:функция-член:put"

```
istream &get(char &ch)
ostream &put(char ch)
```

Функция **get()** считывает отдельный символ из потока и записывает его в переменную *ch*. Кроме того, функция **get()** возвращает ссылку на поток. Функция **put()** записывает переменную *ch* в поток и возвращает ссылку на поток.

Следующая программа отображает на экране содержимое любого файла (как текстового, так и бинарного). Для считывания данных она использует функцию **get()**.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "вызов: PR <имя файла>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Невозможно открыть файл.";
    return 1;
  }

  while(in) { // Если достигнут конец файла,
              // значение объекта in равно false.
    in.get(ch);
    if(in) cout << ch;
  }

  return 0;
}
```

Как указывалось в предыдущем разделе, по достижении конца файла поток, связанный с этим файлом, принимает значение **false**. Следовательно, рано или поздно объект **in** примет значение **false**, и выполнение цикла **while** прекратится.

Приведенный выше цикл можно записать короче.

```
while(in.get(ch))
  cout << ch;
```

Этот фрагмент является правильным, поскольку функция **get()** возвращает ссылку на поток **in**, который примет значение **false** при обнаружении конца файла.

В следующей программе для записи в файл CHARS всех символов от 0 до 255 применяется функция **put()**. Как известно, символы ASCII занимают лишь половину из возможных значений типа **char**. Остальные символы, как правило, называются *расширенными* (extended character set). К ним относятся буквы национальных алфавитов и математические знаки. (Некоторые системы не поддерживают расширенные символы.)

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int i;
  ofstream out("CHARS", ios::out | ios::binary);

  if(!out) {
    cout << "невозможно открыть файл.\n";
```

```
    return 1;
  }

  // Записать символы на диск.
  for(i=0; i<256; i++) out.put((char) i);

  out.close();
  return 0;
}
```

С помощью этой программы можно проверить, поддерживает ли ваш компьютер расширенные символы.

Функции read() и write()

Блоки бинарных данных можно считывать с помощью функций **read()** и **write()**. Их прототипы выглядят следующим образом.

```
istream &read(char *buf, streamsize num);
istream &write(const char *buf, streamsize num);
```

Функция **read** считывает *num* символов из потока и записывает их в буфер, на который ссылается указатель *buf*. Функция **write** записывает *num* символов в поток, считывая их из буфера, на который ссылается указатель *buf*. Как разъяснялось в предыдущей главе, тип **streamsize** определен в библиотеке как разновидность типа **int**. Он позволяет хранить максимальное количество символов, которые могут преобразовываться при выполнении операций ввода-вывода.

Следующая программа записывает структуру на диск, а затем считывает ее обратно.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

struct status {
  char name[80];
  double balance;
  unsigned long account_num;
};

int main()
{
  struct status acc;

  strcpy(acc.name, "Ральф Трантор");
  acc.balance = 1123.23;
  acc.account_num = 34235678;

  // Записываем данные
  ofstream outbal("balance", ios::out | ios::binary);
  if(!outbal) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  outbal.write((char *) &acc, sizeof(struct status));
  outbal.close();
```

```
  // Считываем данные вновь
  ifstream inbal("balance", ios::in | ios::binary);
  if(!inbal) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  inbal.read((char *) &acc, sizeof(struct status));

  cout << acc.name << endl;
  cout << "Счет # " << acc.account_num;
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << endl << "Баланс: $" << acc.balance;

  inbal.close();
  return 0;
}
```

Как видим, для считывания или записи целой структуры достаточно одного вызова функции **read()** или **write()**. Отдельное поле структуры невозможно считать или записать отдельно. Кроме того, этот пример показывает, что буфером может служить объект любого типа.

На заметку

Если буфер не является символьным массивом, при вызове функций **read()** *и* **write()** *необходимо выполнять приведение типов. Поскольку в языке C++ выполняется строгая проверка типов, указатель одного типа не может автоматически преобразовываться в указатель другого типа.*

Если конец файла обнаружится прежде, чем будут считаны *num* символов, функция **read()** просто прекратит работу, а в буфере будет записано максимально возможное количество символов. Количество считанных символов можно определить с помощью функции **gcount()**. Ее прототип выглядит следующим образом.

```
streamsize gcount();
```

Данная функция возвращает количество символов, считанных при выполнении последней операции бинарного ввода. Рассмотрим еще один пример использования функций **read()**, **write()** и **gcount()**.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  double fnum[4] = {99.75, -34.4, 1776.0, 200.1};
  int i;

  ofstream out("numbers", ios::out | ios::binary);
  if(!out) {
    cout << "Невозможно открыть файл.";
    return 1;
   }

  out.write((char *) &fnum, sizeof fnum);

  out.close();
```

```
  for(i=0; i<4; i++) // Очистка массива.
    fnum[i] = 0.0;

  ifstream in("numbers", ios::in | ios::binary);
  in.read((char *) &fnum, sizeof fnum);

  // Определяем количество считанных символов.
  cout << "Считано " << in.gcount() << " байтов.\n";

  for(i=0; i<4; i++) // Показать значения, считанные из файла.
  cout << fnum[i] << " ";

  in.close();

  return 0;
}
```

Эта программа записывает массив десятичных чисел на жесткий диск, а затем считывает их обратно. После вызова функции **read()** применяется функция **gcount()**, возвращающая количество считанных символов.

Дополнительные функции get()

Кроме предыдущих форм функция **get()** имеет еще несколько перегруженных разновидностей. Рассмотрим три основных прототипа.

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get();
```

Первая форма функции **get** считывает символы из массива, на который ссылается указатель *buf*, пока не будут считаны *num*-1 символов, обнаружен символ перехода на следующую строку или достигнут конец файла. Функция **get()** записывает нулевой символ в конец массива, на который ссылается указатель *buf*. Символ перехода на новую строку *не считывается*. Он остается в потоке, пока не будет выполнена следующая операция ввода.

Вторая форма функции **get** считывает символы в массив, на который ссылается указатель *buf*, пока не будут считаны *num*-1 символов, обнаружен символ *delim* или достигнут конец файла. Функция **get()** записывает нулевой символ в конец массива, на который ссылается указатель *buf*. Символ *delim* из потока *не считывается*. Он остается в потоке, пока не будет выполнена следующая операция ввода.

Третья форма функции **get** извлекает из потока следующий символ. Если обнаружен конец файла, она возвращает константу **EOF**. Эта форма функции **get()** похожа на функцию **getc()** в языке C.

Функция getline()

Кроме функции **get()** в языке C++ существует функция **getline()**. Она является членом каждого потокового класса. Ее прототипы выглядят следующим образом.

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

Первая форма функции **getline** считывает символы из массива, на который ссылается указатель *buf*, пока не будут считаны *num*–1 символов, обнаружен символ перехода на следующую строку или достигнут конец файла. Функция **getline()** записывает нулевой символ в конец массива, на который ссылается указатель *buf*. Символ перехода на новую строку извлекается из потока, но *не записывается* в буфер.

Вторая форма функции **getline** считывает символы в массив, на который ссылается указатель *buf*, пока не будут считаны *num*–1 символов, обнаружен символ *delim* или достигнут конец файла. Функция **getline()** записывает нулевой символ в конец массива, на который ссылается указатель *buf*. Символ *delim* извлекается из потока, но не записывается в буфер.

Как видим, две версии функции **getline()** очень напоминают версии **get(buf, num)** и **get(buf, num, delim)**. Они считывают символы из массива, на который ссылается указатель *buf*, пока не будут считаны *num*–1 символов, обнаружен разделитель или достигнут конец файла. Разница заключается в том, что в отличие от функции **get()** функция **getline()** удаляет разделитель из потока.

Рассмотрим программу, демонстрирующую работу функции **getline()**. Она считывает содержимое текстового файла и выводит его на экран.

```
// Программа считывает и выводит на экран
// строки, считанные из текстового файла.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Вызов: Display <имя функции>\n";
    return 1;
  }

  ifstream in(argv[1]); // Ввод

  if(!in) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  char str[255];

  while(in) {
    in.getline(str, 255);  // По умолчанию delim == '\n'.
    if(in) cout << str << endl;
  }

  in.close();

  return 0;
}
```

Распознавание конца файла

Распознать конец файла можно с помощью функции **eof()**, имеющей прототип

```
bool eof();
```

Если достигнут конец файла, она возвращает значение **true**, в противном случае она возвращает значение **false**.

Применение функции **eof()** иллюстрируется следующей программой, которая выводит на экран содержимое файла в шестнадцатеричном формате и в виде ASCII-кодов. (Сохранен текст оригинала. — *Прим. ред.*)

```
/* Display contents of specified file
   in both ASCII and in hex.
*/
#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: Display <filename>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  register int i, j;
  int count = 0;
  char c[16];

  cout.setf(ios::uppercase);
  while(!in.eof()) {
    for(i=0; i<16 && !in.eof(); i++) {
      in.get(c[i]);
    }
    if(i<16) i--; // get rid of eof

    for(j=0; j<i; j++)
      cout << setw(3) << hex << (int) c[j];
    for(; j<16; j++) cout << "   ";

    cout << "\t";
    for(j=0; j<i; j++)
      if(isprint(c[j])) cout << c[j];
      else cout << ".";

    cout << endl;

    count++;
    if(count==16) {
      count = 0;
      cout << "Press ENTER to continue: ";
      cin.get();
      cout << endl;
    }
  }
```

```
  in.close();

  return 0;
}
```

Если в качестве параметра этой программы задать ее исходный файл, на экране появятся следующие результаты.

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65    /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64    nts of specified
20 66 69 6C 65  D  A 20 20 20 69 6E 20 62 6F 74    file..   in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68    h ASCII and in h
65 78 2E  D  A 2A 2F  D  A 23 69 6E 63 6C 75 64    ex...*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 3E  D  A 23 69    e <iostream>..#i
6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61 6D 3E    nclude <fstream>
 D  A 23 69 6E 63 6C 75 64 65 20 3C 63 63 74 79    ..#include <ccty
70 65 3E  D  A 23 69 6E 63 6C 75 64 65 20 3C 69    pe>..#include <i
6F 6D 61 6E 69 70 3E  D  A  D  A 69 6E 74 20 6D    omanip>....int m
61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 63 68    ain(int argc, ch
61 72 20 2A 61 72 67 76 5B 5D 29  D  A 7B  D  A    ar *argv[])..{..
20 20 69 66 28 61 72 67 63 21 3D 32 29 20  D  A    if(argc!=2)  ..
20 20 7B  D  A 20 20 20 20 63 6F 75 74 20 3C 3C    {..     cout <<
20 22 55 73 61 67 65 3A 20 44 69 73 70 6C 61 79    "Usage: Display
20 3C 66 69 6C 65 6E 61 6D 65 3E 5C 6E 22 3B  D    <filename>\n";.
```

Функция ignore()

С помощью функции **ignore()** можно считать и проигнорировать символ из входного потока.

```
istream &ignore(streamsize num=1, int_type delim=EOF);
```

Она считывает и отбрасывает символы, пока не будут пропущены *num* символов (по умолчанию параметр *num* равен 1) или не встретится символ *delim*, который по умолчанию равен константе **EOF**. Обнаруженный разделитель не удаляется из потока ввода. Тип **int_type** определен как разновидность типа **int**.

Следующая программа считывает файл TEST. Она игнорирует символы, пока не встретится пробел или не будут считаны 10 символов. Затем она выводит на экран остальную часть файла.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("test");

  if(!in) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  /* Игнорируются 10 символов, пока не встретится пробел. */
  in.ignore(10, ' ');
  char c;
  while(in) {
```

```
    in.get(c);
    if(in) cout << c;
  }

  in.close();
  return 0;
}
```

Функции peek() и putback()

Можно считать следующий символ из потока, не извлекая его оттуда. Для этого предназначена функция **peek()**, имеющая прототип, приведенный ниже.

```
int_type peek();
```

Эта функция возвращает следующий символ из потока ввода или признак конца файла. Тип **int_type** определен как разновидность типа **int**.

Символ, считанный из потока последним, можно вернуть обратно с помощью функции **putback()**. Ее прототип имеет следующий вид.

```
istream &putback(char c);
```

Здесь параметр *c* означает символ, считанный последним.

Функция flush()

При выводе данные не сразу передаются физическому устройству, связанному с потоком. Вместо этого они накапливаются во внутреннем буфере, пока он не заполнится. Однако существует способ принудительно записать информацию из буфера на диск, не дожидаясь его заполнения. Для этого предназначена функция **flush()**. Ее прототип имеет следующий вид.

```
ostream &flush();
```

Функцию **flush** следует вызывать, когда программа выполняется в неблагоприятных условиях (например, если часто происходят сбои питания).

На заметку *Закрытие файла или прекращение работы программы также очищает все буферы.*

Произвольный доступ

Произвольный доступ к файлу обеспечивается функциями **seekg()** и **seekp()**. Их прототипы имеют следующий вид.

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

Тип **off_type** является разновидностью целого типа. Он определен в классе **ios** и позволяет хранить максимальные значения, которые может принимать параметр *offset*. Тип **seekdir** представляет собой перечисление, определенное в классе **ios**. В нем содержатся разновидности поиска, выполняемого функциями **seekg** и **seekp**.

Система ввода-вывода языка C++ управляет двумя указателями, связанными с файлами. Первый указатель, определяющий позицию, в которой выполняется чтение файла, называется *курсором чтения* (get pointer). Другой указатель, определяющий позицию, в которой выполняется запись в файл, называется *курсором записи* (put pointer). Каждый раз при выполнении ввода и вывода соответствующий курсор файла перемещается на одну позицию вперед. Однако функции **seekg()** и **seekp()** позволяют выполнять произвольные перемещения по файлу.

Функция **seekg()** перемещает связанный с ней курсор записи на *offset* символов, отсчитывая от позиции *origin*. Позиция *origin* задается тремя возможными значениями.

ios::beg	Начало файла
ios::beg	Текущее положение
ios::end	Конец файла

Функция **seekp()** перемещает связанный с ней курсор записи на *offset* символов, отсчитывая от позиции *origin*. Позиция *origin* задается тремя возможными значениями, указанными выше.

Как правило, произвольный доступ при вводе-выводе осуществляется только к файлу, открытому в бинарном режиме. Преобразование символов при считывании текстовых файлов может нарушить правильный порядок следования байтов в файле.

Следующая программа демонстрирует работу функции **seekp()**. Она позволяет изменять указанный символ в файле. В командной строке следует задать имя файла и позицию изменяемого символа, за которой следует новый символ. Обратите внимание на то, что файл открыт для операций ввода-вывода.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Применение: CHANGE <имя файла> <старый символ> <новый символ>\n";
    return 1;
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);
  if(!out) {
    cout << "Невозможно открыть файл.";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);

  out.put(*argv[3]);
  out.close();

  return 0;
}
```

Например, чтобы применить эту программу для замены двенадцатого символа буквой Z, следует выполнить такую команду.

```
change test 12 Z
```

Следующая программа использует функцию **seekg()**. Она выводит на экран содержимое файла, начиная с позиции, указанной в командной строке.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Применение: SHOW <имя файла> <начальная позиция>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Невозможно открыть файл.";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);

  while(in.get(ch))
    cout << ch;

  return 0;
}
```

Ниже показано, как с помощью функций **seekp()** и **seekg()** переставить в обратном порядке первые *num* символов в файле.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Применение: Reverse <имя файла> <num>\n";
    return 1;
  }

  fstream inout(argv[1], ios::in | ios::out | ios::binary);

  if(!inout) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  long e, i, j;
  char c1, c2;
  e = atol(argv[2]);

  for(i=0, j=e; i<j; i++, j--) {
    inout.seekg(i, ios::beg);
    inout.get(c1);
    inout.seekg(j, ios::beg);
    inout.get(c2);
```

```
    inout.seekp(i, ios::beg);
    inout.put(c2);
    inout.seekp(j, ios::beg);
    inout.put(c1);
  }

  inout.close();
  return 0;
}
```

Определение текущей позиции

Текущую позицию курсоров чтения и записи можно определить с помощью функций **tellg()** и **tellp()**. Их прототипы имеют следующий вид.

```
pos_type tellg();
pos_type tellp();
```

Тип **pos_type** определен в классе **ios** и позволяет хранить максимальное значение, которое может вернуть функция. Значения, возвращаемые функциями **tellp()** и **tellg()**, можно использовать в качестве аргументов функции **seekg()** и **seekp()** соответственно.

```
istream &seekg(pos_type pos);
ostream &seekg(pos_type pos);
```

Эти функции позволяют сохранить текущее положение файлового курсора, выполнить определенные файловые операции, а затем восстановить прежнее положение курсора.

Статус ввода-вывода

Система ввода-вывода языка C++ сохраняет информацию о результате каждой операции ввода-вывода. Текущее состояние системы ввода-вывода хранится в объекте класса **iostate**, который является перечислением, определенным в классе **ios**. Кроме этого, класс **ios** содержит следующие члены.

Имя	*Значение*
ios::goodbit	Набор байтов, описывающих нормальное состояние.
ios::eofbit	1, если обнаружен конец файла, 0 в противном случае.
ios::failbit	1, если обнаружена (возможно) поправимая ошибка, 0 в противном случае.
ios::badbit	1, если обнаружена непоправимая ошибка, 0 в противном случае.

Существуют два способа получить информацию о статусе ввода-вывода. Во-первых, можно вызвать функцию **rdstate()**. Она имеет следующий прототип.

```
iostate rdstate();
```

Эта функция возвращает текущее состояние флагов ошибок. Как следует из вышесказанного, если никаких ошибок не обнаружено, функция **rdstate()** возвращает значение **goodbit**. В противном случае устанавливается флаг ошибки.

Применение функции **rdstate()** иллюстрируется следующей программой.

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Применение: Display <имя файла>\n";
    return 1;
  }

  ifstream in(argv[1]);

  if(!in) {
    cout << "Невозможно открыть файл.\n";
    return 1;
  }

  char c;
  while(in.get(c)) {
    if(in) cout << c;
    checkstatus(in);
  }

  checkstatus(in);  // Проверка заключительного состояния.
  in.close();
  return 0;
}

void checkstatus(ifstream &in)
{
  ios::iostate i;

  i = in.rdstate();

  if(i & ios::eofbit)
    cout << "Обнаружен конец файла\n";
  else if(i & ios::failbit)
    cout << "Обнаружена поправимая ошибка\n";
  else if(i & ios::badbit)
    cout << "Обнаружена непоправимая ошибка\n";
}
```

Эта программа всегда обнаруживает одну "ошибку". После завершения цикла **while** последний вызов функции **checkstatus()**, как и ожидалось, обнаруживает конец файла. Эта функция может оказаться полезной в любой программе.

Второй способ обнаружения ошибки основан на применении следующих функций.

```
bool bad();
bool eof();
bool fail();
bool good();
```

Функция **bad()** возвращает значение **true**, если установлен флаг **badbit**. Функция **eof()** возвращает значение **true**, если установлен флаг **failbit**. Функция **good()** возвращает значение **true**, если никаких ошибок не обнаружено. В противном случае функция возвращает значение **false**.

Флаги, соответствующие обнаруженным ошибкам, можно сбросить. Для этого следует вызвать **clear()**, прототип которой имеет следующий вид.

```
void clear(iostate flags=ios::goodbit);
```

Если параметр *flags* является объектом **goodbit** (по умолчанию), все флаги ошибок сбрасываются. В противном случае параметр *flags* следует задать произвольно.

Настройка ввода-вывода в файлы

В главе 20 были описаны операторы вставки и извлечения, перегруженные для собственных классов. В этой главе был рассмотрен лишь консольный ввод-вывод. Однако, поскольку все потоки в языке C++ одинаковы, перегрузку операторов вставки и извлечения можно применять как для консоли, так и для файлов. В качестве примера проанализируем следующую программу, которая модифицирует электронную телефонную книгу, описанную в главе 20. Эта программа весьма проста. Она позволяет добавлять имена в список или выводить список имен на экран. Для ввода и вывода телефонных номеров применяются перегруженные функции вставки и извлечения. Вы можете переделать эту программу, дополнив ее возможностями поиска нужного номера или удаления нежелательных номеров.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class phonebook {
  char name[80];
  char areacode[4];
  char prefix[4];
  char num[5];
public:
  phonebook() { };
  phonebook(char *n, char *a, char *p, char *nm)
  {
    strcpy(name, n);
    strcpy(areacode, a);
    strcpy(prefix, p);
    strcpy(num, nm);
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Выводит на экран имя и номер телефона.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-";
  stream << o.num << "\n";
  return stream; // Функция должна возвращать ссылку на поток.
}

// Вводит имя и номер телефона.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Введите имя: ";
  stream >> o.name;
```

```
  cout << "Введите код города: ";
  stream >> o.areacode;
  cout << "Введите префикс: ";
  stream >> o.prefix;
  cout << "Введите номер: ";
  stream >> o.num;
  cout << "\n";
  return stream;
}

int main()
{
  phonebook a;
  char c;

  fstream pb("phone", ios::in | ios::out | ios::app);

  if(!pb) {
    cout << "Невозможно открыть файл с телефонной книгой.\n";
    return 1;
  }

  for(;;) {
    do {
      cout << "1. Ввод номеров\n";
      cout << "2. Вывод номеров\n";
      cout << "3. Выход\n";
      cout << "\nВыберите пункт меню: ";
      cin >> c;
    } while(c<'1' || c>'3');

    switch(c) {
      case '1':
        cin >> a;
        cout << "Выбранный пункт: ";
        cout << a;  // Вывод на экран.
        pb << a;  // Запись на диск.
        break;
      case '2':
        char ch;
        pb.seekg(0, ios::beg);
        while(!pb.eof()) {
          pb.get(ch);
          if(!pb.eof()) cout << ch;
        }
        pb.clear();  // Сброс признака конца файла.
        cout << endl;
        break;
      case '3':
        pb.close();
        return 0;
    }
  }
}
```

Обратите внимание на то, что оператор "<<" теперь можно использовать как для записи данных в файл на диске, так и для вывода информации на экран без каких-либо изменений. Это одно из наиболее важных и полезных свойств системы ввода-вывода языка C++.

Полный справочник по

Глава 22

Динамическая идентификация типа и операторы приведения

В языке C++ для поддержки объектно-ориентированного программирования используется динамическая идентификация типа (RTTI — Run-Time Type Identification) и четыре дополнительных оператора приведения типов. Ни одно из этих свойств не упоминалось в исходной версии языка C++. Они были добавлены впоследствии для поддержки динамического полиморфизма. Система RTTI позволяет идентифицировать тип объекта при выполнении программы. Дополнительные операторы приведения типов обеспечивают более безопасный способ приведения. Поскольку один из этих операторов приведения (оператор **dynamic_cast**) непосредственно связан с системой динамической идентификации типов, имеет смысл рассмотреть их в одной главе.

Динамическая идентификация типа (RTTI)

Динамическая идентификация типов не характерна для таких неполиморфных языков, как язык C. В этих языках нет необходимости определять тип объекта при выполнении программы, поскольку тип каждого объекта известен еще на этапе компиляции. Однако в полиморфных языках, таких как C++, возникают ситуации, в которых тип объекта на этапе компиляции неизвестен, поскольку природа этого объекта уточняется только в ходе выполнения программы. Как описано в главе 17, язык C++ реализует полиморфизм с помощью иерархий классов, виртуальных функций и указателей на объекты базового класса. Поскольку указатели на объекты базового класса могут ссылаться и на объекты *производных классов*, не всегда можно предсказать, на объект какого типа они будут ссылаться в тот или иной момент. Эту идентификацию приходится осуществлять в ходе выполнения программы.

Для идентификации типа объекта используется оператор **typeid**, определенный в заголовке **<typeinfo>**. Чаще всего он применяется в следующей форме.

```
typeid(object)
```

Здесь параметр *object* является объектом, тип которого мы хотим идентифицировать. Он может иметь любой тип, в том числе встроенный или определенный пользователем. Оператор **typeid** возвращает ссылку на объект типа **type_info**, описывающий тип *объекта*.

Класс **type_info** содержит следующие открытые функции-члены.

```
bool operator==(const type_info &ob);
bool operator!=(const type_info &ob);
bool before(const type_info &ob);
const char *name();
```

Перегруженные операторы "**==**" и "**!=**" позволяют сравнивать типы. Если вызывающий объект предшествует объекту, использованному как параметр, функция **before()** возвращает значение **true**. (Чаще всего эта функция применяется для внутренних целей. Она не имеет никакого отношения к иерархии классов и механизму наследования.) Функция **name()** возвращает указатель на имя заданного типа.

Рассмотрим пример, в котором используется оператор **typeid**.

```
// Простой пример, в котором используется оператор typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class myclass1 {
  // ...
};
```

```
class myclass2 {
  // ...
};

int main()
{
  int i, j;
  float f;
  char *p;
  myclass1 ob1;
  myclass2 ob2;

  cout << "Тип объекта i: " << typeid(i).name();
  cout << endl;
  cout << "Тип объекта f: " << typeid(f).name();
  cout << endl;
  cout << "Тип объекта p: " << typeid(p).name();
  cout << endl;

  cout << "Тип объекта ob1: " << typeid(ob1).name();
  cout << endl;
  cout << "Тип объекта ob2: " << typeid(ob2).name();
  cout << "\n\n";

  if(typeid(i) == typeid(j))
    cout << "Типы объектов i и j совпадают\n";

  if(typeid(i) != typeid(f))
    cout << "Типы объектов i и f не совпадают\n";

  if(typeid(ob1) != typeid(ob2))
    cout << "Объекты ob1 и ob2 имеют разные типы\n";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Тип объекта i: int
Тип объекта f: float
Тип объекта p: char *
Тип объекта ob1: class myclass1
Тип объекта ob2: class myclass2

Типы объектов i и j совпадают
Типы объектов i и f не совпадают
Объекты ob1 и ob2 имеют разные типы
```

Самое важное свойство оператора **typeid** проявляется, когда он применяется к указателю на объект базового класса. В этом случае он автоматически возвращает тип реального объекта, на который ссылается указатель, причем этот объект может быть экземпляром как базового, так и производного классов. (Напомним еще раз: указатель на объект базового класса может ссылаться на объекты производного класса.) Таким образом, в ходе выполнения программы, используя оператор **typeid**, можно идентифицировать тип объекта, на который ссылается указатель базового класса. Этот принцип иллюстрируется следующей программой.

```
// Пример применения оператора typeid
// к иерархии полиморфных классов.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Класс Mammal
                                             // является полиморфным
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

int main()
{
  Mammal *p, AnyMammal;
  Cat cat;
  Platypus platypus;

  p = &AnyMammal;
  cout << "Указатель p ссылается на объект типа ";
  cout << typeid(*p).name() << endl;

  p = &cat;
  cout << "Указатель p ссылается на объект типа ";
  cout << typeid(*p).name() << endl;

  p = &platypus;
  cout << "Указатель p ссылается на объект типа ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Cat
Указатель p ссылается на объект типа class Platypus
```

Итак, если оператор **typeid** применяется к указателю на объект полиморфного базового класса, тип объекта, на который ссылается указатель, можно определить в ходе выполнения программы.

В любом случае, если оператор **typeid** применяется к указателю на объект неполиморфного базового класса, возвращается базовый тип указателя. Иначе говоря, невозможно определить, на объект какого типа ссылается этот указатель на самом деле.

Закомментируем, например, ключевое слово **virtual**, стоящее перед именем функции **lays_egg()** в классе **Mammal**, перекомпилируем программу и запустим ее вновь.

```
Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal
```

Класс **Mammal** больше не является полиморфным, и каждый объект теперь имеет тип **Mammal**, т.е. тип указателя **p**.

Поскольку оператор **typeid** обычно применяется к разыменованным указателям, следует предусмотреть обработку исключительной ситуации **bad_typeid**, которая может возникнуть, если указатель будет нулевым.

Ссылки на объект полиморфного класса аналогичны указателям. Если оператор **typeid** применяется к ссылке на объект полиморфного базового класса, он возвращает тип объекта, на который в действительности установлена ссылка, причем этот тип может быть производным классом. Чаще всего этот оператор применяют для проверки параметров, передаваемых функциям с помощью ссылок. Например, в следующей программе функция **WhatMammal()** объявляет параметр, являющийся ссылкой на объект класса **Mammal**. Это значит, что функция **WhatMammal()** может получать ссылки на объекты класса **Mammal**, а также любого класса, производного от него. Если к этому параметру применяется оператор **typeid**, он возвращает фактический тип передаваемого объекта.

```
// Оператор typeid применяется к ссылке.
#include <iostream>
#include <typeinfo>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Класс Mammal
                                             // является полиморфным
  // ...
};

class Cat: public Mammal {
public:
  // ...
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  // ...
};

// Применение оператора typeid к параметру,
// передаваемому с помощью ссылки
void WhatMammal(Mammal &ob)
{
  cout << "Объект ob имеет тип ";
  cout << typeid(ob).name() << endl;
}

int main()
{
  Mammal AnyMammal;
```

```
  Cat cat;
  Platypus platypus;

  WhatMammal(AnyMammal);
  WhatMammal(cat);
  WhatMammal(platypus);

  return 0;
}
```

Эта программа выводит на экран следующие результаты.

```
Объект ob имеет тип class Mammal
Объект ob имеет тип class Cat
Объект ob имеет тип class Platypus
```

Оператор **typeid** имеет вторую форму, получающую в качестве аргумента имя типа. Ее общий вид приведен ниже.

```
typeid(имя_типа)
```

Например, следующий оператор совершенно правилен

```
cout << typeid(int).name();
```

Данная форма оператора **typeid** позволяет получить объект класса **type_info**, описывающий заданный тип. Благодаря этому ее можно применять в операторах сравнения типов. Например, следующая версия функции **WhatMammal()** сообщает, что кошки боятся воды.

```
void WhatMammal(Mammal &ob)
{
  cout << "Параметр ob ссылается на объект типа ";
  cout << typeid(ob).name() << endl;
  if(typeid(ob) == typeid(Cat))
    cout << "Кошки боятся воды.\n";
}
```

Применение динамической идентификации типа

Мощь механизма динамической идентификации типа иллюстрируется следующей программой. Функция **factory()** создает объекты различных классов, производных от класса **Mammal**. (Функции, создающие объекты, иногда называются *фабриками объектов* (object factory).) Конкретный тип создаваемого объекта задается функцией **rand()**, которая представляет собой генератор случайных чисел в языке C++. Таким образом, тип создаваемого объекта заранее не известен. Программа создаст 10 объектов и подсчитывает количество объектов каждого типа. Поскольку все объекты генерируются функцией **factory()**, для идентификации фактического типа объекта используется оператор **typeid**.

```
// Демонстрация динамической идентификации типа.
#include <iostream>
using namespace std;

class Mammal {
public:
  virtual bool lays_eggs() { return false; } // Класс Mammal
                                             // является полиморфным
  // ...
};
```

```
class Cat: public Mammal {
public:
  //  ..
};

class Platypus: public Mammal {
public:
  bool lays_eggs() { return true; }
  //  ..
};

class Dog: public Mammal {
public:
  // ...
};

// Фабрика объектов, производных от класса Mammal.
Mammal *factory()
{
  switch(rand() % 3 ) {
    case 0: return new Dog;
    case 1: return new Cat;
    case 2: return new Platypus;
  }
  return 0;
}

int main()
{
  Mammal *ptr; // Указатель на базовый класс.
  int i;
  int c=0, d=0, p=0;

  // Создаем и подсчитываем объекты.
  for(i=0; i<10; i++) {
    ptr = factory(); // Создаем объект.

    cout << "Тип объекта: " << typeid(*ptr).name();
    cout << endl;

    // Подсчет
    if(typeid(*ptr) == typeid(Dog)) d++;
    if(typeid(*ptr) == typeid(Cat)) c++;
    if(typeid(*ptr) == typeid(Platypus)) p++;
  }

  cout << endl;
  cout << "Созданные животные:\n";
  cout << "  Собаки: " << d << endl;
  cout << "  Кошки: " << c << endl;
  cout << "  Утконосы: " << p << endl;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Тип объекта: class Platypus
Тип объекта: class Platypus
Тип объекта: class Cat
Тип объекта: class Cat
Тип объекта: class Platypus
Тип объекта: class Cat
Тип объекта: class Dog
Тип объекта: class Dos
Тип объекта: class Cat
Тип объекта: class Platypus

Созданные животные:
  Собаки: 2
  Кошки: 4
  Утконосы: 4
```

Применение оператора typeid к шаблонным классам

Оператор **typeid** можно применять к шаблонным классам. Тип объекта, являющегося объектом шаблонного класса, определяется, в частности, тем, какие данные используются в качестве обобщенных при создании конкретного объекта. Если при создании двух экземпляров шаблонного класса используются данные разных типов, считается, что эти объекты имеют разные типы. Рассмотрим простой пример.

```
// Применение оператора typeid к шаблонам.
#include <iostream>
using namespace std;

template <class T> class myclass {
  T a;
public:
  myclass(T i) { a = i; }
  // ...
};

int main()
{
  myclass<int> o1(10), o2(9);
  myclass<double> o3(7.2);

  cout << "Объект o1 имеет тип ";
  cout << typeid(o1).name() << endl;

  cout << "Объект o2 имеет вид ";
  cout << typeid(o2).name() << endl;

  cout << "Объект o3 имеет вид ";
  cout << typeid(o3).name() << endl;

  cout << endl;

  if(typeid(o1) == typeid(o2))
    cout << "Объекты o1 и o2 имеют одинаковые типы\n";

  if(typeid(o1) == typeid(o3))
    cout << "Ошибка\n";
  else
    cout << "Объекты o1 и o3 имеют разные типы\n";
```

```
  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Объект o1 имеет тип class myclass<int>
Объект o2 имеет тип class myclass<int>
Объект o3 имеет тип class myclass<double>

Объекты o1 и o2 имеют разные типы
Объекты o1 и o3 имеют разные типы
```

Как видим, хотя два объекта представляют собой экземпляры одного и того же шаблонного класса, если параметры не совпадают, их типы считаются разными. В данной программе объект **o1** имеет тип **myclass<int>**, а объект **o3** — **myclass<double>**. Таким образом, их типы не совпадают.

Динамическая идентификация типов применяется не во всех программах. Однако при работе с полиморфными типами механизм RTTI позволяет распознавать типы объектов в любых ситуациях.

Операторы приведения типов

В языке C++ существуют пять операторов приведения типов. Первый оператор является вполне традиционным и унаследован от языка C. Остальные четыре были добавлены впоследствии. К ним относятся операторы **dynamic_cast**, **const_cast**, **reinterpret_cast** и **static_cast**. Эти операторы позволяют полнее контролировать процессы приведения типов.

Оператор dynamic_cast

Пожалуй, наиболее важным из перечисленных новшеств является оператор **dynamic_cast**, осуществляющий динамическое приведение типа с последующей проверкой корректности приведения. Если приведение оказалось некорректным, оно не выполняется. Общий вид оператора **dynamic_cast** таков.

```
dynamic_cast <target_type> (expr);
```

Здесь параметр *target_type* задает результирующий тип, а параметр *expr* — выражение, которое приводится к новому типу. Результирующий тип должен быть указательным или ссылочным, а приводимое выражение — вычислять указатель или ссылку. Таким образом, оператор **dynamic_cast** можно применять для приведения типов указателей или ссылок.

Оператор **dynamic_cast** предназначен для приведения полиморфных типов. Допустим, даны два полиморфных класса **B** и **D**, причем класс **D** является производным от класса **B**. Тогда оператор **dynamic_cast** может привести указатель типа **D*** к типу **B***. Это возможно благодаря тому, что указатель на объект базового класса может ссылаться на объект производного класса. Однако обратное динамическое приведение указателя типа **D*** к типу **B*** возможно лишь в том случае, если указатель действительно ссылается на объект класса **D**. Оператор **dynamic_cast** достигает цели, если указатель или ссылка, подлежавшие приведению, ссылаются на объект результирующего класса или объект класса, производного от результирующего. В противном случае приведение типов считается неудавшимся. В случае неудачи

оператор **dynamic_cast**, примененный к указателям, возвращает нулевой указатель. Если оператор **dynamic_cast** применяется к ссылкам, в случае ошибки генерируется исключительная ситуация **bad_cast**.

Рассмотрим простой пример. Допустим, что класс **Base** является полиморфным, а **Derived** — производным от него.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // Указатель базового типа ссылается на объект
            // производного класса.
dp = dynamic_cast<Derived *> (bp); // Приведение указателя
            // производного типа выполнено успешно.
if(dp) cout << "Приведение выполнено успешно";
```

Приведение указателя **bp**, имеющего базовый тип, к указателю **dp**, имеющему производный тип, выполняется успешно, поскольку указатель **bp** на самом деле ссылается на объект класса **Derived**. Таким образом, этот фрагмент выводит на экран сообщение "**Приведение выполнено успешно**". Однако в следующем фрагменте приведение не выполняется, потому что указатель **bp** ссылается на объект класса **Base**, а приведение базового объекта к производному невозможно.

```
bp = &b_ob; // Указатель базового типа ссылается на объект
            // класса Base.
dp = dynamic_cast<Derived *> (bp); // Ошибка
if(!dp) cout << "Приведение не выполняется";
```

Поскольку приведение невозможно, фрагмент выводит на экран сообщение "**Приведение не выполняется**".

Следующая программа демонстрирует разные ситуации, в которых выполняется оператор **dynamic_cast**.

```
// Демонстрация оператора dynamic_cast.
#include <iostream>
using namespace std;

class Base {
public:
  virtual void f() { cout << "Внутри класса Base\n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "Внутри класса Derived\n"; }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Приведение типа Derived * к типу Derived * "
         << "выполнено успешно.\n";
```

```
  dp->f();
} else
  cout << "Ошибка\n";

cout << endl;

bp = dynamic_cast<Base *> (&d_ob);
if(bp) {
  cout << "Приведение типа Derived * к типу Base * "
       << "выполнено успешно.\n";
  bp->f();
} else
  cout << "Ошибка\n";

cout << endl;

bp = dynamic_cast<Base *> (&b_ob);
if(bp) {
  cout << "Приведение типа Base * к типу Base * "
          "выполнено успешно.\n";
  bp->f();
} else
  cout << "Ошибка\n";

cout << endl;

dp = dynamic_cast<Derived *> (&b_ob);
if(dp)
  cout << "Ошибка\n";
else
  cout << "Приведение типа Base * к типу Derived * "
          "невозможно.\n";

cout << endl;

bp = &d_ob; // Указатель bp ссылается на объект
            // класса Derived.
dp = dynamic_cast<Derived *> (bp);
if(dp) {
  cout << "Приведение указателя bp к типу Derived * "
       << "выполнено успешно,\n" <<
       << "поскольку bp действительно ссылается\n"
       << "на объект класса Derived.\n";
  dp->f();
} else
  cout << "Ошибка\n";

cout << endl;

bp = &b_ob; // Указатель bp ссылается на объект класса Base.
dp = dynamic_cast<Derived *> (bp);
if(dp)
  cout << "Ошибка";
else {
  cout << "Теперь приведение указателя bp к типу Derived *\n"
       << "невозможно, так как указатель bp на самом деле \n"
       << "ссылается на объект класса Base.\n";
}
```

```
  cout << endl;

  dp = &d_ob; // Указатель dp ссылается на объект класса Derived
  bp = dynamic_cast<Base *> (dp);
  if(bp) {
    cout << "Приведение указателя dp к типу Base *"
         << "выполнено успешно.\n";
    bp->f();
  } else
    cout << "Ошибка\n";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Приведение типа Derived * к типу Derived * выполнено успешно.
Внутри класса Derived

Приведение типа Derived * к типу Base * выполнено успешно.
Внутри класса Derived

Приведение типа Base * к типу Base * выполнено успешно.
Внутри класса Base

Приведение типа Base * к типу Derived * невозможно.

Приведение указателя bp к классу Derived * выполнено успешно,
поскольку указатель bp действительно ссылается
на объект класса Derived
Внутри класса Derived

Теперь приведение указателя bp к типу Derived
невозможно, так как указатель bp на самом деле
ссылается на объект класса Base.

Приведение указателя dp к классу Base * выполнено успешно.
Внутри класса Derived
```

Замена оператора typeid оператором dynamic_cast

Иногда оператор **dynamic_cast** можно использовать вместо оператора **typeid**. Допустим, что класс **Base** является полиморфным, а **Derived** — производным от него. В следующем фрагменте указателю **dp** присваивается адрес объекта, на который ссылается указатель **bp**, только если этот объект действительно является экземпляром класса **Derived**.

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

В этом фрагменте используется традиционный оператор приведения. Это вполне безопасно, поскольку оператор **if** проверяет легальность приведения с помощью оператора **typeid**. Однако в этой ситуации лучше заменить оператор **typeid** и условный оператор **if** оператором **dynamic_cast**.

```
dp = dynamic_cast<Derived *> (bp);
```

Поскольку оператор **dynamic_cast** выполняется успешно, только если приводимый объект является либо экземпляром результирующего класса, либо объектом класса, производного от результирующего, указатель **dp** содержит либо нулевой указатель, либо адрес объектов типа **Derived**. Поскольку оператор **dynamic_cast** выполняется успешно, только если приведение является легальным, его применение иногда упрощает ситуацию. Замену оператора **typeid** оператором **dynamic_cast** иллюстрирует следующая программа. В ней одна и та же операция выполняется дважды: сначала — с помощью оператора **typeid**, а затем — оператора **dynamic_cast**.

```
// Замена оператора typeid оператором dynamic_cast.
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
public:
  virtual void f() {}
};

class Derived : public Base {
public:
  void derivedOnly() {
    cout << "Объект класса Derived.\n";
  }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  // ***********************************
  // Применение оператора typeid
  // ***********************************
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Приведение типа Base к типу Derived невозможно.\n";

  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else
    cout << "Ошибка, приведение невозможно!\n";

  // ***********************************
  // Применение оператора dynamic_cast
  // ***********************************
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
```

```
    cout << "Приведение типа Base к типу Derived невозможно.\n";

  bp = &d_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else
    cout << "Ошибка, приведение невозможно!\n";

  return 0;
}
```

Как видим, оператор **dynamic_cast** упрощает логику приведения указателя базового типа к указателю производного типа.

```
Приведение типа Base к типу Derived невозможно.
Объект класса Derived.
Приведение типа Base к типу Derived невозможно.
Объект класса Derived.
```

Применение оператора dynamic_cast к шаблонным классам

Оператор **dynamic_cast** можно применять к шаблонным классам. Рассмотрим пример.

```
// Применение операторв dynamic_cast к шаблонным классам.
#include <iostream>
using namespace std;

template <class T> class Num {
protected:
  T val;
public:
  Num(T x) { val = x; }
  virtual T getval() { return val; }
  // ...
};

template <class T> class SqrNum : public Num<T> {
public:
  SqrNum(T x) : Num<T>(x) { }
  T getval() { return val * val; }
};

int main()
{
  Num<int> *bp, numInt_ob(2);
  SqrNum<int> *dp, sqrInt_ob(3);
  Num<double> numDouble_ob(3.3);

  bp = dynamic_cast<Num<int> *> (&sqrInt_ob);
  if(bp) {
    cout << "Приведение типа SqrNum<int>* к типу Num<int>*"
         << "выполнено успешно.\n";
    cout << "Значение равно " << bp->getval() << endl;
  } else
    cout << "Ошибка\n";

  cout << endl;
```

```
  dp = dynamic_cast<SqrNum<int> *> (&numInt_ob);
  if(dp)
    cout << "Ошибка\n";
  else {
    cout << "Приведение типа Num<int>* к типу SqrNum<int>*"
         << "невозможно.\n";
    cout << "Приведение указателя на объект базового класса\n";
    cout << "к указателю на объект производного класса невозможно.\n";
  }
  cout << endl;

  bp = dynamic_cast<Num<int> *> (&numDouble_ob);
  if(bp)
    cout << "Ошибка\n";
  else
    cout << "Приведение типа Num<double>* к типу Num<int>* невозможно.\n";
    cout << "Это два разных типа.\n";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Приведение типа SqrNum<int>* к типу Num<int>* выполнено успешно.
Значение равно 9

Приведение типа Num<int>* к типу SqrNum<int>* невозможно.
Приведение указателя на объект базового класса
к указателю на объект производного класса невозможно.

Приведение типа Num<double>* к типу Num<int>* невозможно.
Это два разных типа.
```

Основной смысл этого фрагмента заключается в том, что с помощью оператора **dynamic_cast** нельзя привести указатель на объект одной шаблонной специализации к указателю на экземпляр другой шаблонной специализации. Напомним, что точный тип объекта шаблонного класса определяется типом данных, которые используются при его создании. Таким образом, типы **Num<double>** и **Num<int>** различаются.

Оператор const_cast

Оператор **const_cast** используется для явного замещения модификаторов **const** и/или **volatile**. Результирующий тип должен совпадать с исходным, за исключением атрибутов **const** и **volatile**. Чаще всего оператор **const_cast** применяется для удаления модификатора **const**. Общий вид оператора **const_cast** приведен ниже.

```
const_cast<type>(expr)
```

Здесь параметр *type* задает результирующий тип приведения, а параметр *expr* является выражением, которое приводится к новому типу.

Рассмотрим программу, демонстрирующую применение оператора **const_cast**.

```
// Демонстрация оператора const_cast.
#include <iostream>
using namespace std;

void sqrval(const int *val)
```

```
{
  int *p;

  // Удаление модификатора const.
  p = const_cast<int *> (val);

  *p = *val * *val;
}

int main()
{
  int x = 10;

  cout << "Значение x перед вызовом: " << x << endl;
  sqrval(&x);
  cout << "Значение x после вызова: " << x << endl;

  return 0;
}
```

Эта программа выводит на экран следующие результаты.

```
Значение x перед вызовом: 10
Значение x после вызова: 100
```

Как видим, функция **sqrval()** изменяет значение переменной **x**, даже если ее параметр является константным указателем.

Кроме того, оператор **const_cast** можно использовать для удаления модификатора **const** у константной ссылки. Например, предыдущую программу можно изменить так, чтобы параметр функции был константной ссылкой.

```
// Применение оператора const_cast к константной ссылке.
#include <iostream>
using namespace std;

void sqrval(const int &val)
{
  // Удаление модификатора const у ссылки val.
  const_cast<int &> (val) = val * val;
}

int main()
{
  int x = 10;

  cout << "Значение x перед вызовом: " << x << endl;
  sqrval(x);
  cout << "Значение x после вызова: " << x << endl;

  return 0;
}
```

Программа вычисляет те же результаты, что и в предыдущем случае, и работает лишь благодаря тому, что оператор **const_cast** временно удаляет атрибут **const** у ссылки **val**, позволяя присвоить ей новое значение (в данном случае значение переменной **x**).

Следует подчеркнуть, что применение оператора **const_cast** для удаления атрибута **const** небезопасно. Его следует использовать очень осторожно.

И последнее: атрибут **const** можно удалить только с помощью оператора **const_cast**. Операторы **dynamic_cast**, **static_cast** и **reinterpret_cast** на атрибут **const** не влияют.

Оператор static_cast

Оператор **static_cast** выполняет неполиморфное приведение. Его можно применять для любого стандартного преобразования типов. Проверка приведения в ходе выполнения программы не производится. Оператор **static_cast** имеет следующий вид.

```
static_cast<type>(expr)
```

Здесь параметр *type* задает результирующий тип приведения, а параметр *expr* — выражение, которое приводится к новому типу.

Оператор **static_cast**, по существу, заменяет исходный оператор приведения. Просто он выполняет неполиморфное приведение. Например, следующая программа приводит переменную типа **int** к типу **double**.

```
// Применение оператора static_cast.
#include <iostream>
using namespace std;

int main()
{
  int i;

  for(i=0; i<10; i++)
    cout << static_cast<double> (i) / 3 << " ";

  return 0;
}
```

Оператор reinterpret_cast

Оператор **reinterpret_cast** преобразует один тип в совершенно другой. Например, он может преобразовать указатель на целое число в целое число, а целое число — в указатель. Кроме того, его можно использовать для приведения несовместимых типов указателей. Оператор **reinterpret_cast** имеет следующий вид.

```
reinterpret_cast<type>(expr)
```

Здесь параметр *type* задает результирующий тип приведения, а параметр *expr* — выражение, которое приводится к новому типу.

Рассмотрим программу, иллюстрирующую применение оператора **reinterpret_cast**.

```
// Пример, в котором используется оператор reinterpret_cast.
#include <iostream>
using namespace std;

int main()
{
  int i;
  char *p = "Это строка";

  i = reinterpret_cast<int> (p); // Приведение указателя
                                 // к целому числу.
```

```
    cout << i;

    return 0;
}
```

Здесь оператор **reinterpret_cast** преобразует указатель **p** в целое число, т.е. один основной тип — в другой. Такое преобразование является типичным для оператора **reinterpret_cast**.

Полный справочник по

Глава 23

Пространства имен, преобразования функций и другие новшества

В этой главе описываются пространства имен и другие новшества, включая функции преобразования, явные конструкторы, функции-члены с атрибутами **const** и **volatile**, ключевое слово **asm** и спецификации связей. Она завершается обсуждением буферизованного ввода-вывода и перечислением различий между языками C и C++.

Пространства имен

Понятие пространства имен появилось относительно недавно. Пространства имен предназначены для локализации имен идентификаторов и предотвращения их конфликтов. Среда программирования C++ наполнена большим количеством переменных, функций и классов. Раньше все их имена пребывали в глобальном пространстве и нередко конфликтовали между собой. Например, если в программе определена функция **abs()**, она может замещать собой стандартную функцию **abs()**, поскольку имена обеих функций находятся в глобальном пространстве имен. Чаще всего конфликты имен возникали, когда программа использовала несколько сторонних библиотек одновременно. Особенно это касается имен классов. Например, если в программе был определен класс **ThreeDCircle**, а в библиотеке, которую использует эта программа, имя уже было задействовано, возникал конфликт.

Введение ключевого слова **namespace** позволило решить эти проблемы. Поскольку пространство имен позволяет локализовать область видимости объектов, объявленных внутри него, одно и то же имя, упомянутое в разных контекстах, больше не вызывает конфликтов. Наибольшую пользу это нововведение принесло стандартной библиотеке языка C++. До сих пор вся стандартная библиотека языка C++ находилась в глобальном пространстве имен (которое, собственно говоря, было единственным). Теперь стандартная библиотека определена внутри своего собственного пространства имен **std**, что намного уменьшает вероятность конфликтов. Программист может создавать свои собственные пространства имен и самостоятельно локализовать имена, которые могут вызывать конфликты. Это особенно важно при разработке классов или библиотек функций.

Основы пространств имен

Ключевое слово **namespace** позволяет разделить глобальное пространство имен на декларативные области (declarative region). В сущности, пространство имен — это область видимости. Общий вид объявления пространства имен таков.

```
namespace name
{
   // Объявления
}
```

Все, что объявлено в разделе **namespace**, находится внутри области видимости этого пространства имен.

Рассмотрим пример пространства имен. Оно локализует имена, использованные при создании простого класса, реализующего обратный счетчик. В этом пространстве имен определен класс **counter** и переменные **upperbound** и **lowerbound**, содержащие верхнюю и нижнюю границу диапазона счетчика.

```
namespace CounterNameSpace
{
  int upperbound;
  int lowerbound;
```

```
  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n)
     {
       if(n <= upperbound) count = n;
     }

     int run()
     {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}
```

Здесь переменные **upperbound** и **lowerbound**, а также класс **counter** находятся в области видимости, определенной пространством имен **CounterNameSpace**.

К идентификаторам, объявленным внутри пространства имен, можно обращаться непосредственно, не указывая квалификатор. Например, внутри пространства имен **CounterNameSpace** функция **run()** может прямо ссылаться на переменную **lowerbound**.

```
if(count > lowerbound) return count--;
```

Однако, поскольку пространство имен определяет область видимости, для ссылок на объекты, находящиеся вне этого пространства, необходимо применять оператор разрешения области видимости. Например, чтобы присвоить число 10 переменной **upperbound** в модуле, находящемся вне пространства имен **CounterNameSpace**, следует выполнить оператор, приведенный ниже.

```
CounterNameSpace::upperbound = 10;
```

Чтобы объявить объект класса **counter** вне пространства имен **CounterNameSpace**, необходимо применить следующий оператор.

```
CounterNameSpace::counter ob;
```

Как правило, чтобы обратиться к элементу пространства имен извне, следует перед его именем указать имя пространства и оператор разрешения области видимости.

Рассмотрим программу, демонстрирующую применение пространства имен **CounterNameSpace**.

```
// Демонстрация пространства имен.
#include <iostream>
using namespace std;

namespace CounterNameSpace
{
  int upperbound;
  int lowerbound;

  class counter {
     int count;
```

```
  public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }

    void reset(int n)
    {
      if(n <= upperbound) count = n;
    }

    int run()
    {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
  };
}

int main()
{
  CounterNameSpace::upperbound = 100;
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  CounterNameSpace::counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  ob2.reset(100);
  CounterNameSpace::lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);

  return 0;
}
```

Обратите внимание на то, что объявлению объекта класса **counter** и ссылкам на переменные **upperbound** и **lowerbound** предшествует квалификатор **CounterNameSpace**. Однако после объявления объекта класса **counter** его члены можно использовать без квалификатора. Таким образом, функцию **ob1.run()** можно вызывать непосредственно, поскольку соответствующее пространство имен подразумевается.

Директива using

Легко представить, во что превратится программа, в которой часто встречаются ссылки на элементы пространства имен. К сожалению, ее текст станет малопонятным, поскольку будет пестреть квалификаторами и операторами области разрешения видимости. Чтобы избежать этого, следует применять директиву **using**, имеющую такой вид.

```
using namespace name;
using name::member;
```

В первом варианте параметр *name* задает название пространства имен. Все элементы этого пространства становятся частью текущей области видимости и могут использоваться без указания квалификатора. Во втором варианте в область видимости включается только конкретный элемент пространства имен. Предположим, что в программе объявлено пространство имен **CounterNameSpace**. Тогда можно применить следующие операторы.

```
using CounterNameSpace::lowerbound; // Видимым является
                                    // только член lowerbound.
lowerbound = 10; // Правильный оператор, поскольку переменная
                 // lowerbound является видимой.

using namespace CounterNameSpace; // Все члены видимы.
upperbound = 100; // Правильный оператор, поскольку все члены
                  // являются видимыми.
```

Следующая программа представляет собой переработанный вариант предыдущей программы, иллюстрирующий применение оператора **using**.

```
// Демонстрация директивы using.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n)
     {
       if(n <= upperbound) count = n;
     }

     int run()
     {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}

int main()
{
```

```
  // Используется только член upperbound
  // из пространства имен CounterNameSpace.
  using CounterNameSpace::upperbound;

  // Теперь квалификатор перед переменной upperbound не нужен.
  upperbound = 100;

  // Перед переменной lowerbound квалификатор нужен.
  CounterNameSpace::lowerbound = 0;

  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  } while(i > CounterNameSpace::lowerbound);
  cout << endl;

  // Теперь доступно все пространство имен CounterNameSpace.
  using namespace CounterNameSpace;

  counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);
  cout << endl;

  ob2.reset(100);
  lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  } while(i > lowerbound);

  return 0;
}
```

Эта программа иллюстрирует еще одно полезное свойство: одно пространство имен не перекрывает другого. Открывая пространство имен, вы просто добавляете его члены к элементам текущей области видимости. Таким образом, в конце программы, кроме глобального пространства имен, доступными оказываются также пространства имен **std** и **CounterNameSpace**.

Неименованные пространства имен

Существует особая разновидность пространств имен — *неименованное пространство имен* (unnamed namespace), позволяющее создавать уникальные идентификаторы, область видимости которых ограничена файлом. Неименованные пространства имен иногда называют *безымянными* (anonymous namespaces). Их объявление имеет следующий вид.

```
namespace
{
   // Объявления
}
```

Неименованные пространства имен позволяют создавать уникальные идентификаторы, область видимости которых ограничена файлом. Иначе говоря, внутри файла, содержащего неименованное пространство имен, элементы этого пространства можно использовать без квалификаторов. Вне файла эти переменные считаются невидимыми.

Неименованные пространства имен позволяют избежать применения спецификатора **static**. Как указывалось в главе 2, спецификатор **static** позволяет сузить область видимости глобального имени до размера файла. Рассмотрим два файла, являющихся частью одной и той же программы.

Первый файл	*Второй файл*

`static int k;`
`void f1()`
`{`
`  k = 99; // Все правильно!`
`}`
	`extern int k;`
	`void f2()`
	`{`
	`  k = 10; // Ошибка!`
	`}`

Поскольку переменная **k** определена в первом файле, в нем она используется без каких-либо ограничений. Во втором файле переменная **k** определена с помощью ключевого слова **extern**. Это означает, что в данном файле имя и тип переменной **k** считаются известными, хотя сама она в нем не определена. Если эти два файла связаны между собой, попытка использовать переменную **k** во втором файле вызовет ошибку, поскольку переменная **k** объявлена статической, и ее область видимости ограничена первым файлом.

Несмотря на то что язык C++ по-прежнему допускает применение статических глобальных переменных, того же эффекта можно добиться с помощью неименованного пространства имен. Рассмотрим пример.

Первый файл	*Второй файл*

`namespace`
`{`
`  int k;`
`}`
`void f1()`
`{`
`  k = 99; // Все правильно!`
`}`
	`extern int k;`
	`void f2()`
	`{`
	`  k = 10; // Ошибка!`
	`}`

В этом примере область видимости переменной **k** по-прежнему ограничена первым файлом. Создавая новые программы, вместо спецификатора **static** следует применять неименованные пространства имен.

Некоторые особенности пространств имен

Одно и то же пространство имен можно объявлять несколько раз. Это позволяет распределить одно пространство имен среди нескольких файлов и даже разделить его внутри одного файла.

```
#include <iostream>
using namespace std;

namespace NS {
  int i;
}

// ...
```

```
namespace NS {
  int j;
}

int main()
{
  NS::i = NS::j = 10;

  // Конкретная ссылка на пространство имен NS.
  cout << NS::i * NS::j << "\n";

  // Применение пространства NS.
  using namespace NS;

  cout << i * j;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
100
100
```

В данном примере пространство имен **NS** разделено на две части. Однако содержимое каждой части по-прежнему находится в одном и том же пространстве имен **NS**.

Пространство имен должно быть объявлено вне какой бы то ни было области видимости. Это значит, что пространство имен нельзя объявлять, например, внутри функции. Однако существует одно исключение: пространства имен могут быть вложенными. Рассмотрим следующую программу.

```
#include <iostream>
using namespace std;

namespace NS1 {
  int i;
  namespace NS2
  { // Вложенное пространство имен
    int j;
  }
}

int main()
{
  NS1::i = 19;
  // NS2::j = 10; Ошибка, пространство имен NS2
  // находится вне области видимости.
  NS1::NS2::j = 10; // Это правильно!

  cout << NS1::i << " "<<  NS1::NS2::j << "\n";

  // Использование пространства имен NS1
  using namespace NS1;

  /* Теперь, поскольку пространство имен NS1 находится
     в области видимости, для ссылки на переменную j
     можно использовать пространство имен NS2. */
  cout << i * NS2::j;
```

```
  return 0;
}
```

Результаты этой программы приведены ниже.

```
19 10
190
```

В данном случае пространство имен **NS2** вложено в пространство имен **NS1**. Следовательно, в начале программы при ссылке на переменную **j** нужно указывать названия обоих пространств имен, которым она принадлежит. Одного имени **NS2** недостаточно. После выполнения директивы

```
using namespace NS1;
```

можно указывать только имя **NS2**, поскольку пространство имен **NS1** уже находится в области видимости.

Обычно в небольших программах нет необходимости создавать пространства имен. Однако библиотеки, которые должны быть как можно более независимыми, следует помещать внутрь собственного пространства имен.

Пространство имен std

Стандартная библиотека языка C++ пребывает в собственном пространстве имен **std**. По этой причине большинство программ, приведенных в книге, содержат директиву

```
using namespace std;
```

Этот оператор открывает прямой доступ к функциям и классам, определенным внутри библиотеки, поэтому квалификатор **std::** не нужен.

Разумеется, при желании можно явно указывать квалификатор **std::**. Например, в следующей программе стандартная библиотека не включается в глобальное пространство имен.

```
// Применение явного квалификатора std::.
#include <iostream>

int main()
{
  int val;

  std::cout << "Введите номер: ";

  std::cin >> val;

  std::cout << "Это ваш номер: ";
  std::cout << std::hex << val;

  return 0;
}
```

Здесь потоки **cout**, **cin** и манипулятор **hex** сопровождаются указанием квалификатора **std::**. Иначе говоря, при записи в стандартный поток вывода нужно задавать имя **std::cout**, при чтении из стандартного потока ввода следует задавать имя **std::cin**, а манипулятор формата необходимо вызывать по имени **std::hex**.

Если программа не очень широко использует средства стандартной библиотеки, ее можно не включать в глобальное пространство имен. Однако, если программа содер-

жит сотни обращений к стандартным функциям и классам, будет слишком утомительно каждый раз указывать квалификатор **std::**.

Если в программе используется лишь несколько членов стандартной библиотеки, имеет смысл применить директиву **using** к каждому из них отдельно. Эти имена по-прежнему можно использовать, не указывая спецификатор **std::**, однако при этом вся стандартная библиотека не погружается в глобальное пространство имен. Рассмотрим пример.

```
// В глобальное пространство имен включается
// лишь несколько идентификаторов.
#include <iostream>

// Предоставляем доступ к именам cout, cin и hex.
using std::cout;
using std::cin;
using std::hex;

int main()
{
  int val;

  cout << "Введите номер: ";

  cin >> val;
  cout << "Это ваш номер: ";
  cout << hex << val;
  return 0;
}
```

В этой программе имена **cin**, **cout** и **hex** можно использовать непосредственно, а остальная часть библиотеки **std** остается вне области видимости.

Как известно, первоначально библиотека языка C++ находилась в глобальном пространстве имен. Если вы работаете со старыми программами на языке C++, в них необходимо включить оператор

```
using namespace std;
```

или указывать перед каждым членом библиотеки квалификатор **std::**. Это особенно важно, если старые заголовочные файлы заменяются новыми. Помните, что старые заголовочные файлы **(.H)** помещают свое содержимое в глобальное пространство имен, а новые — в пространство имен **std**.

Создание функций преобразования

В некоторых ситуациях объект некоего класса должен входить в выражение, содержащее данные других типов. Иногда этого можно достичь с помощью перегруженных операторных функций. Однако в других случаях необходимо простое преобразование типа в более широкий тип. Для этого язык C++ позволяет создавать собственные *функции преобразования* (conversion functions). Функция преобразования трансформирует класс в другой тип, совместимый с остальной частью выражения. Общий вид функции преобразования таков.

```
operator type() { return value;}
```

Здесь параметр *type* задает результирующий тип, в который преобразуется класс, а параметр *value* определяет значение объекта после преобразования. Функция пре-

образования возвращает объект типа *type*. Никакой другой спецификатор типа не допускается. Кроме того, функция не имеет аргументов. Функция преобразования должна быть членом класса, для которого она определена. Функции преобразования наследуются и могут быть виртуальными.

Рассмотрим преобразование класса **stack**, описанного в главе 11. Допустим, что объект класса **stack** необходимо использовать внутри целочисленного выражения. Кроме того, будем считать, что значением объекта класса **stack**, использованного в целочисленном выражении, является количество элементов, хранящихся в стеке. (Иногда в процессе моделирования или мониторинга необходимо отслеживать скорость заполнения стека.) Для этого достаточно преобразовать стек в целое число, равное количеству его элементов. Чтобы сделать это, можно применить следующую функцию преобразования.

```
operator int() { return tos; }
```

Рассмотрим программу, иллюстрирующую работу функции преобразования.

```
#include <iostream>
using namespace std;

const int SIZE=100;

// Создается класс stack
class stack {
  int stck[SIZE];
  int tos;
public:
  stack() { tos=0; }
  void push(int i);
  int pop(void);
  operator int() { return tos; } // Преобразование стека
                                 // в целое число.
};

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Стек полон.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Стек пуст.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack stck;
  int i, j;
```

```
  for(i=0; i<20; i++)  stck.push(i);

  j = stck; // Преобразование в целое число.

  cout << j << " элементов в стеке.\n";

  cout << SIZE - stck << " свободных мест.\n";
  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
20 элементов в стеке.
80 свободных мест.
```

Как видим, когда объект класса **stack** используется в целочисленном выражении, например **j=stck**, к нему применяется функция преобразования. В данном случае функция преобразования возвращает число 20. Кроме того, функция преобразования вызывается при вычитании объекта **stck** из константы **SIZE**.

Рассмотрим еще один пример функции преобразования. Эта программа создает класс **pwr**, который хранит и вычисляет степень некоторого числа. Результат записывается в переменную типа **double**. Применяя функцию преобразования к объекту класса **pwr**, можно использовать его в выражении типа **double**.

```
#include <iostream>
using namespace std;

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  pwr operator+(pwr o) {
    double base;
    int exp;
    base = b + o.b;
    exp = e + o.e;

    pwr temp(base, exp);
    return temp;
  }
  operator double() { return val; } // Преобразование в тип double
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
}

int main()
{
  pwr x(4.0, 2);
  double a;
```

```
  a = x; // Преобразование в тип double.
  cout << x + 100.2; // Преобразуем x в тип double
                     // и добавляем число 100.2.
  cout << "\n";

  pwr y(3.3, 3), z(0, 0);

  z = x + y;  // Преобразование не производится.
  a = z;  // Преобразование в тип double.
  cout << a;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
116.2
20730.7
```

Как видим, когда объект **x** используется в выражении **x+100.2**, вызывается функция преобразования, создающая значение типа **double**. Обратите внимание на то, что в выражении **x+y** преобразование типа не вызывается, поскольку оба операнда имеют тип **pwr**.

Из предыдущих примеров следует, что функция преобразования оказывается полезной во многих ситуациях. Довольно часто функции преобразования позволяют создавать более естественные синтаксические конструкции смешанных выражений, в которые входят как переменные встроенных типов, так и объекты классов, определенных пользователем. В частности, на примере класса **pwr** видно, что применение функции преобразования в тип **double** позволяет использовать объекты в обычных математических выражениях.

В программе можно создать несколько функций преобразования для трансформации объектов в разные типы. Например, в предыдущий пример можно было бы добавить функцию, преобразовывающую объект класса **pwr** в тип **long**. Каждая функция преобразования автоматически применяется к объекту, входящему в выражение соответствующего типа.

Функции-члены с атрибутами const и mutable

Функции — члены класса можно объявлять константными. В этом случае указатель **this** считается константным. Следовательно, такая функция не может модифицировать вызывающий ее объект. Однако константную функцию-член могут применять как константные, так и неконстантные объекты.

Чтобы определить константную функцию-член, следует использовать следующую форму объявления.

```
class X {
  int some_var;
public:
  int f1() const; // Константная функция-член
};
```

Как видим, атрибут **const** указывается после объявления параметров функции.

Функцию-член объявляют константной, чтобы запретить ей модифицировать вызывающий объект. Рассмотрим следующий пример.

```
/*
   Демонстрация константной функции-члена.
```

```
   Эта программа не компилируется.
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // OK!
  }

  void seti(int x) const {
    i = x; // Ошибка!
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

Эта программа не компилируется, поскольку функция-член **seti()** объявлена константной. Это значит, что она не может модифицировать вызывающий объект. При попытке изменить значение переменной **i** возникнет ошибка. В противоположность этому функция **geti()** не пытается модифицировать переменную **i**, что вполне допускается.

Иногда возникает необходимость модифицировать тот или иной член класса с помощью константной функции-члена, сохраняя в неприкосновенности остальную часть. Для этого предназначено ключевое слово **mutable**. Оно отменяет атрибут **const**. Иначе говоря, член, объявленный с атрибутом **mutable**, может модифицироваться константной функцией-членом. Рассмотрим пример.

```
// Демонстрация ключевого слова mutable.
#include <iostream>
using namespace std;

class Demo {
  mutable int i;
  int j;
public:
  int geti() const {
    return i; // Все верно!
  }

  void seti(int x) const {
    i = x; // Теперь все верно!
  }

/* Эта функция не компилируется.
  void setj(int x) const {
    j = x; // По-прежнему неверно!
  }
*/
};
```

```
int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

Здесь переменная **i** объявлена с атрибутом **mutable**, поэтому функция **seti()** может ее модифицировать. Однако переменная **j** атрибутом **mutable** не обладает, поэтому функция **setj()** не имеет права изменять ее значение.

Функции-члены с атрибутом volatile

Функции—члены класса могут иметь атрибут **volatile**. Следовательно, в этом случае указатель **this** также имеет атрибут **volatile**. Чтобы определить функцию-член с атрибутом **volatile**, необходимо использовать следующую форму объявления.

```
class X {
public:
  void f2(int a) volatile; // Функция-член с атрибутом volatile.
};
```

Явные конструкторы

Как указывалось в главе 12, если конструктор имеет только один аргумент, для инициализации объекта *ob* можно применять выражения *ob(x)* или *ob=x*. Причина заключается в том, что одновременно с созданием конструктора, имеющего один аргумент, неявно создается функция преобразования аргумента в объект данного класса. Однако иногда автоматическое преобразование аргумента оказывается нежелательным. Для этой цели в языке C++ предусмотрено ключевое слово **explicit**. Рассмотрим следующий пример.

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
  myclass ob = 4; // Автоматически преобразуется
                  // в выражение  myclass(4).

  cout << ob.geta();
```

```
  return 0;
}
```

Здесь конструктор **myclass** имеет один параметр. Обратите особое внимание на объявление объекта **ob** в функции **main()**. Оператор

```
myclass ob = 4;
```

автоматически преобразуется в вызов конструктора **myclass()** с параметром 4. Иначе говоря, компилятор не отличает его от оператора

```
myclass ob(4);
```

Чтобы избежать неявного преобразования аргумента, можно использовать ключевое слово **explicit**. Спецификатор **explicit** применяется только к конструкторам. Конструктор с атрибутом **explicit** используется только при обычном вызове. Он не выполняет никаких автоматических преобразований. Например, если конструктор **myclass()** обладает атрибутом **explicit**, автоматическое преобразование его аргумента не производится. Рассмотрим вариант, в котором объявлен явный конструктор класса **myclass**.

```
#include <iostream>
using namespace std;

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

Теперь возможен только один способ вызова конструктора.

```
myclass ob(4);
```

Инициализация

```
myclass ob = 4; // Ошибка
```

теперь невозможна.

Инициализация членов класса

В предыдущих главах мы рассматривали инициализацию членов класса внутри конструкторов. Например, следующая программа содержит класс **MyClass** с двумя целочисленными членами **numA** и **numB**, которые инициализируются внутри конструктора.

```
#include <iostream>
using namespace std;

class MyClass {
  int numA;
  int numB;
public:
  /* Члены numA и numB инициализируются внутри конструктора
     с помощью обычной синтаксической конструкции.
  */
  MyClass(int x, int y) {
```

```
    numA = x;
    numB = y;
  }

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Значения в объекте ob1 равны " << ob1.getNumB()
       << " и " << ob1.getNumA() <<endl;

  cout << "Значения в объекте ob2 равны " << ob2.getNumB()
       << " и " << ob2.getNumA() <<endl;

  return 0;
}
```

В этом примере переменные-члены **numA** и **numB** инициализируются внутри конструктора класса **MyClass** как обычно. Однако такой подход оказывается приемлемым не во всех случаях. Например, если бы переменные **numA** и **numB** были константными, объявление класса выглядело бы так.

```
class MyClass {
  const int numA; // Константный член
  const int numB; // Константный член
```

В этом случае данным переменным ничего нельзя присвоить. Аналогичные проблемы возникают, если членом класса является ссылка, которую необходимо инициализировать, и если в классе не предусмотрен конструктор по умолчанию. Чтобы решить эту проблему, в языке C++ был предложен альтернативный способ инициализации членов класса при создании объекта.

Синтаксическая конструкция, предназначенная для инициализации членов класса, похожа на конструктор базового класса. Она имеет следующий вид.

```
конструктор(список_аргументов) :member1(инициализатор),
                                member2(инициализатор),
                                  // ...
                                memberN(инициализатор)
{
   // Тело конструктора
}
```

Члены класса, подлежащие инициализации, перечисляются перед телом конструктора и отделяются двоеточием от имени конструктора и списка аргументов. Инициализацию членов класса и вызовы конструкторов базового класса можно смешивать в одном выражении.

Перепишем класс **MyClass**, сделав члены **numA** и **numB** константными. Инициализация этих членов выполняется с помощью списка инициализации.

```
#include <iostream>
using namespace std;

class MyClass {
  const int numA; // Константный член
```

```
  const int numB; // Константный член
public:
  /* Члены numA и numB инициализируются внутри конструктора
     с помощью списка инициализации.
  */
  MyClass(intx, int y):numA(x), numB(y) {}

  int getNumA() { return numA; }
  int getNumB() { return numB; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Значения в объекте ob1 равны " << ob1.getNumB()
       << " и " << ob1.getNumA() <<endl;

  cout << "Значения в объекте ob2 равны " << ob2.getNumB()
       << " и " << ob2.getNumA() <<endl;

  return 0;
}
```

Обратите внимание на то, что члены **numA** и **numB** инициализируются оператором

```
MyClass(intx, int y):numA(x), numB(y) {}
```

Здесь переменная **numA** инициализируется значением аргумента **x**, а **numB** — значением переменной **y**. Хотя члены **numA** и **numB** имеют атрибут **const**, при создании объекта класса **MyClass** им присваиваются начальные значения **x** и **y**, поскольку для этого используется список инициализации.

Такой способ инициализации особенно полезен, если член класса имеет тип, для которого не предусмотрен конструктор по умолчанию. Рассмотрим еще одну версию класса **myClass**, в котором два целочисленных значения хранятся в объекте класса **IntPair**.

```
// Эта программа содержит ошибку и не компилируется.

#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) {}
}

class MyClass {
  IntPair nums; // Ошибка: класс IntPair не содержит
                // конструктор по умолчанию.
public:
  // Этот фрагмент не работает!

  MyClass(intx, int y)  {
     nums.a = x;
     nums.b = y;
  }
```

```
  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Значения в объекте ob1 равны " << ob1.getNumB()
       << " и " << ob1.getNumA() <<endl;

  cout << "Значения в объекте ob2 равны " << ob2.getNumB()
       << " и " << ob2.getNumA() <<endl;

  return 0;
}
```

Эта программа не компилируется, так как класс **IntPair** содержит лишь один конструктор и имеет два аргумента. Однако объект **nums** в классе **MyClass** объявлен без параметров, а значения переменных **a** и **b** задаются в конструкторе класса **MyClass**. Это порождает ошибку, поскольку предполагается, что при создании объекта класса **IntPair** должен быть доступен конструктор по умолчанию (т.е. конструктор без параметров), а это условие не выполнено.

Чтобы решить эту проблему, в класс **IntPair** следует добавить конструктор по умолчанию. Однако это возможно лишь тогда, когда доступ к исходному коду класса полностью открыт, а это условие выполняется не всегда. В таких случаях следует применять список инициализации членов класса, как показано ниже.

```
// Исправленная версия программы.

#include <iostream>
using namespace std;

class IntPair {
public:
  int a;
  int b;

  IntPair(int i, int j) : a(i), b(j) {}
}

class MyClass {
  IntPair nums; // Теперь можно!
public:
  // Применение списка инициализации.
  MyClass(intx, int y) : nums(x,y) {}

  int getNumA() { return nums.a; }
  int getNumB() { return nums.b; }
};

int main()
{
  MyClass ob1(7, 9), ob2(5, 2);

  cout << "Значения в объекте ob1 равны " << ob1.getNumB()
       << " и " << ob1.getNumA() <<endl;
```

```
  cout << "Значения в объекте ob2 равны " << ob2.getNumB()
       << " и " << ob2.getNumA() <<endl;

  return 0;
}
```

Теперь при создании объекта класса **MyClass** инициализируется объект **nums**. Таким образом, конструктор по умолчанию не нужен.

И последнее: члены класса создаются и инициализируются в порядке своего объявления в классе, а не в порядке их перечисления в списке инициализации.

Применение ключевого слова asm

Несмотря на всю мощь и выразительность языка C++, в некоторых ситуациях его недостаточно. (Например, в языке C++ нет оператора, который блокировал бы прерывания.) Для особых ситуаций в языке C++ предусмотрели "лазейку", позволяющую вставлять в программу ассемблерный код, игнорируя компилятор. Этой "лазейкой" является оператор **asm**. Используя его, в программу можно включать фрагменты ассемблерного кода, которые при компиляции остаются без изменения, а затем становятся частью выполняемого модуля.

Оператор **asm** имеет следующий вид.

```
asm("code");
```

Параметр *code* представляет собой инструкцию на языке ассемблера, которая вставляется в программу. Однако некоторые компиляторы позволяют также использовать иные формы оператора **asm**.

```
asm инструкция;
asm инструкция переход_на_новую_строку;
asm
{
   последовательность инструкций
}
```

В этих выражениях могут использоваться любые допустимые инструкции на языке ассемблера. Поскольку реализация этого свойства зависит от компилятора, детали следует искать в документации.

В момент написания этой книги в компиляторе Microsoft Visual C++ для включения ассемблерного кода использовалось ключевое слово **__asm**, а в других компиляторах — **asm**.

Рассмотрим пример.

```
#include <iostream>
using namespace std;

int main()
{
  asm int 5; // Генерируется прерывание номер 5.

  return 0;
}
```

При запуске этой программы под управлением системы DOS генерируется инструкция **INT 5**, которая вызывает функцию печати экрана (print-screen).

На заметку *Применение ключевого слова требует хорошего знания языка ассемблера. Если вы не полностью овладели этим языком, не рискуйте, поскольку вставка ошибочного ассемблерного кода может закончиться плачевно.*

Спецификации связей

В языке C++ можно указать, как функции связываются с программой. По умолчанию все функции связываются как функции на языке C++. Однако, используя *спецификации связей* (linkage specification), с программой можно связать функции, написанные на другом языке. Спецификация связей имеет следующий вид.

```
extern "language" прототип_функции
```

Параметр *language* задает требуемый язык. Все компиляторы помогают связывать функции на языке C++ и C. Кроме того, некоторые компиляторы позволяют подключать функции, написанные на языках Fortran, Pascal или BASIC. (Детали изложены в документации, сопровождающей компилятор.)

Следующая программа связывается функцией **myCfunc()**, написанной на языке C.

```
#include <iostream>
using namespace std;

extern "C" void myCfunc();

int main()
{
  myCfunc();

  return 0;
}

// Функция на языке C.
void myCfunc()
{
  cout << "Функция на языке C.\n";
}
```

На заметку *Ключевое слово* ***extern*** *является неотъемлемой частью спецификации связей. Кроме того, спецификация связей должна быть глобальной: ее нельзя применять внутри функции.*

С программой можно связать несколько функций, написанных на одном и том же языке. Для этого используется следующая форма спецификации.

```
extern "language"
{
   прототипы
}
```

Буферизованный ввод-вывод

Кроме консольного и файлового ввода-вывода в языке C++ возможен *буферизованный вывод* (array-base I/O). При этом в качестве устройства ввода, вывода или ввода-вывода используется символьный массив. Буферизованный вывод осуществляется

с помощью обычных потоков. На буферизованный ввод-вывод распространяются практически все ограничения, присущие обычной системе ввода-вывода. Однако существует особенность, делающая систему буферизованного ввода-вывода уникальной — с потоком связывается массив символов. В таком случае говорят, что поток имеет тип **char *.** Для того чтобы иметь возможность использовать потоки типа **char *,** необходимо включить в программу заголовок **<strstream>**.

На заметку *Символьные потоки, описанные в этом разделе, в стандарте языка C++ объявлены нежелательными. Это означает, что их поддержке сохраняется, но их не рекомендуется применять при разработке новых программ. Мы описываем их для программистов, работающих со стерыми программами.*

Классы буферизованного вывода

Существуют три класса буферизованного вывода: **istrstream**, **ostrstream** и **strstream**. Эти классы используются для создания потоков ввода, вывода и ввода-вывода. Кроме того, класс **istrstream** является производным от класса **istream**, класс **ostrstream** — наследником **ostream**, а **strstream** — производным от класса **iostream**. Следовательно, все буферизованные классы являются косвенными наследниками класса **ios** и имеют доступ к тем же самым функциям-членам, что и обычные системы ввода-вывода.

Создание буферизованного потока вывода

Для вывода в массив следует связать массив с потоком, используя конструктор класса **ostrstream**.

```
ostrstream ostr(char *buf, streamsize size, openmode mode=ios::out);
```

Здесь параметр *buf* является указателем на массив, в котором накапливаются символы, записанные в поток *ostr*. Размер этого массива задается параметром *size*. По умолчанию поток открывается для текстового вывода, однако с помощью логической операции "ИЛИ" режим его открытия можно изменять. Например, для параметра *mode* можно использовать значение **ios::app**, позволяющее дописывать информацию в конец массива. Во многих случаях значение параметра *mode* задают по умолчанию.

Если открыт буферизованный поток, весь вывод направляется в массив, связанный с этим потоком. Однако символы не должны переполнять массив, иначе возникнет ошибка.

Рассмотрим простую программу, иллюстрирующую буферизованный вывод.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "Буферизованный вывод в языке C++. ";
  outs << 1024 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ' ' << 99.789 << ends;
```

```
  cout << str;  // Строка выводится на консоль.

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Буферизованный вывод в языке C++. 1024 0x64 99.789
```

Учтите, что поток **outs** практически ничем не отличается от обычных потоков. Единственное отличие заключается в том, что устройством вывода-вывода, связанным с потоком **outs**, является массив символов. Поскольку класс **outs** является потоком, к нему можно применять манипуляторы, например **hex** и **ends**. Кроме того, можно вызывать функции — члены класса **ostream**, например **setf()**.

В следующей программе для дописывания нулевого байта в конец массива применяется манипулятор **ends**. Дописывание может выполняться как автоматически, так и вручную. Это зависит от способа реализации, поэтому стоит не полагаться на компилятор, а самому предусмотреть явную запись нулевого байта в конец массива.

Количество символов, записываемых в массив вывода, можно задать с помощью функции-члена **pcount()**. Ее прототип выглядит так.

```
streamsize pcount();
```

Эта функция учитывает нулевой байт в конце массива.

Проиллюстрируем работу функции **pcount()** следующей программой. Она сообщает, что массив **outs** содержит 18 символов: 17 символов и нулевой байт.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char str[80];

  ostrstream outs(str, sizeof(str));

  outs << "abcdefg ";
  outs << 27 << " "  << 890.23;
  outs << ends;  // Запись нулевого байта.

  cout << outs.pcount(); // Выводим на экран длину массива outs

  cout << " " << str;

  return 0;
}
```

Применение буферизованного ввода

Чтобы связать поток ввода с массивом, следует применять конструктор **istrstream**.

```
istrstream istr(const char *buf);
```

Здесь параметр *buf* является указателем на массив, который используется в качестве источника символов при вводе из потока *istr*. Содержимое массива, на который ссылается указатель *buf*, должно завершаться нулевым байтом. Однако нулевой байт никогда не считывается из массива.

Рассмотрим пример, в котором используется ввод из строки.

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10 Привет 0x75 42.73 OK";

  istrstream ins(s);

  int i;
  char str[80];
  float f;

  // Считываем: 10 Привет
  ins >> i;
  ins >> str;
  cout << i << " " << str << endl;

  // Считываем 0x75 42.73 OK
  ins >> hex >> i;
  ins >> f;
  ins >> str;

  cout << hex << i << " " << f << " " << str;

  return 0;
}
```

Если вводу подлежит лишь часть строки, используется следующая форма конструктора класса **istrstream**.

```
istrstream istr(const char *buf, streamsize size);
```

Здесь используются лишь *size* элементов массива, на который ссылается указатель *buf*. Эта строка не обязана завершаться нулевым байтом, поскольку размер строки определяется параметром *size*.

Потоки, связанные с памятью, ничем не отличаются от потоков, связанных с другими устройствами. Например, следующая программа показывает, как считывается содержимое любого символьного массива. По достижении конца файла объект **ins** принимает значение **false**.

```
/* Программа показывает, как считывается содержимое
   любого символьного массива */
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10.23 это проверка <<>><<?!\n";

  istrstream ins(s);

  char ch;

  /* Считывание и отображение текста на экране. */
```

```
  ins.unsetf(ios::skipws); // Пробелы не пропускать!
  while (ins)
  { // По достижении конца файла объект ins
    // принимает ложное значение.
    ins >> ch;
    cout << ch;
  }

  return 0;
}
```

Буферизованный ввод-вывод

Для создания буферизованного потока, предназначенного для ввода-вывода, используется следующая форма конструктора **strstream**.

```
strstream iostr(char *buf, streamsize size, openmode mode=
                                          ios::in | ios::out);
```

Здесь параметр *buf* является указателем на массив, который используется при выполнении операций ввода-вывода. Размер массива определяется параметром *size*. Значение параметра *mode* определяет режим работы потока *iostr*. Для текстового ввода-вывода используются значения **ios::in | ios::out**. При вводе массив должен заканчиваться нулевым символом.

Рассмотрим программу, в которой для ввода и вывода используется массив.

```
// Операции ввода и вывода.
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char iostr[80];

  strstream strio(iostr, sizeof(iostr), ios::in | ios::out);

  int a, b;
  char str[80];

  strio << "10 20 проверка ";
  strio >> a >> b >> str;
  cout << a << " " << b << " " << str << endl;

  return 0;
}
```

Эта программа сначала записывает строку **10 20 проверка**, а затем считывает ее обратно.

Применение динамических массивов

В предыдущих примерах при связывании потока вывода с массивом его размер передавался с помощью параметра конструктора класса **ostrstream**. Если при выводе размер массива можно предсказать, этот подход вполне оправдан. А если нет? Тогда следует применять следующую форму конструктора класса **ostrstream**.

```
ostrstream();
```

При вызове этого конструктора создается объект класса **ostrstream** и динамический массив, размер которого увеличивается автоматически.

Для доступа к элементам динамического массива используется функция **str()**, имеющая прототип

```
char *str();
```

Эта функция "замораживает" массив и возвращает указатель, который используется для доступа к его элементам. Поскольку динамический массив "заморожен", его нельзя использовать для повторного вывода, пока он не будет "разморожен". Следовательно, массив нельзя "замораживать" в процессе вывода.

Рассмотрим демонстрационную программу.

```
#include <strstream>
#include <iostream>
using namespace std;

int main()
{
  char *p;

  ostrstream outs;  // Динамический массив

  outs << "Буферизованный вывод ";
  outs << -10 << hex << " ";
  outs.setf(ios::showbase);
  outs << 100 << ends;

  p = outs.str(); // Замораживает динамический буфер
                  // и возвращает указатель на него.


  cout << p;

  return 0;
}
```

Динамические массивы можно применять вместе с классом **strstream**, который обеспечивает ввод и вывод.

"Замораживать" и "размораживать" массив можно с помощью функции **freeze()**. Ее прототип показан ниже.

```
void freeze(bool action = true);
```

Если параметр *action* имеет значение **true**, массив "замораживается", если **false** — "размораживается".

Применение бинарных операций ввода-вывода к буферизованным потокам

Напомним, что функциональные возможности буферизованной и обычной системы ввода-вывода совпадают. Следовательно, массивы, связанные с буферизованными потоками, могут содержать бинарную информацию. При считывании бинарной информации для распознавания конца массива используется функция **eof()**. Например, следующая программа демонстрирует, как с помощью функции **getf()** считывается содержимое произвольного массива — бинарного или текстового.

```
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char *p = "Это проверка \1\2\3\4\5\6\7";

  istrstream ins(p);

  char ch;

  // Считываем и выводим на экран бинарную информацию.
  while (!ins.eof()) {
    ins.get(ch);
    cout << hex << (int) ch << ' ';
  }
  return 0;
}
```

В этом примере значения, формируемые символами \1\2\3 и так далее, являются непечатаемыми.

Для вывода бинарных символов используется функция **put()**. Если необходимо считать буфер бинарных данных, можно применить функцию-член **read()**. Для записи буферов бинарных данных следует применять функцию **write()**.

Отличия между языками C и C++

Язык C является подмножеством стандарта языка C++, и практически все программы на языке C можно считать программами на языке C++. И все же между этими языками есть различия, которые уже упоминались в первой и второй частях книги. Перечислим наиболее важные из них.

Локальные переменные в языке C++ можно объявлять в любом месте блока. В языке C эти переменные должны быть объявлены в начале блока, перед первым выполняемым оператором. (В языке C99 это ограничение снято.)

В языке C объявление функции

```
int f();
```

не содержит никакой информации о ее параметрах. Иначе говоря, если список параметров пуст, ничего определенного сказать нельзя. Она может иметь параметры, а может их не иметь. Однако в языке C++ это объявление означает, что функция *не имеет* никаких параметров. Иными словами, в языке C++ следующие два объявления эквивалентны.

```
int f();
int f(void);
```

Параметр **void** необязателен в языке C++. Многие программисты указывают его, чтобы подчеркнуть, что функция действительно не имеет параметров, хотя это совершенно не обязательно.

В языке C++ все функции должны иметь прототипы. В языке C это не обязательно (хотя и свидетельствует о хорошем стиле программирования).

Очень важно, что в языке C символьная константа автоматически преобразуется в целую величину, а в языке C++ — нет.

В языке C глобальную переменную можно объявлять несколько раз, хотя это и дурной тон. В языке C++ это является ошибкой.

Идентификатор в языке C может иметь до 31 символа. В языке C++ все символы являются значимыми. Однако с практической точки зрения очень длинные идентификаторы используются крайне редко.

В языке C внутри программы можно вызывать функцию `main()` (хотя это весьма необычно), а в языке C этого делать нельзя.

Регистровые переменные в языке C не имеют адресов. В языке C++ это возможно.

Если в программе на языке C при объявлении переменной не указан никакой тип, по умолчанию подразумевается тип `int`. Это правило в языке C++ отменено. (В стандарте C99 это правило также не выполняется.)

Полный справочник по

Глава 24

Введение в стандартную библиотеку шаблонов

В этой главе исследуется одна из самых важных составляющих языка C++ — *стандартная библиотека шаблонов* (STL). Ее включение в язык C++ было одним из основных успехов, достигнутых в ходе стандартизации. Библиотека содержит универсальные шаблонные классы и функции, реализующие большое количество широко распространенных алгоритмов и структур данных. Например, в ней определены шаблонные классы векторов, списков, очередей и стеков, а также многочисленные процедуры для работы с ними. Поскольку библиотека STL состоит из шаблонных классов, ее алгоритмы и структуры данных можно применять практически к любым типам данных.

С точки зрения разработки программного обеспечения библиотека стандартных шаблонов представляет собой весьма сложный проект, в котором использованы наиболее изощренные свойства языка C++. Для того чтобы понять и правильно использовать эту библиотеку, необходимо полностью изучить язык C++ и в совершенстве разбираться в указателях, ссылках и шаблонах. Честно говоря, синтаксис шаблонных классов, описанных в библиотеке STL, выглядит довольно устрашающе, хотя на самом деле все не так сложно, как кажется на первый взгляд. Материал, изложенный в этой главе, не сложнее тем, описанных в других частях книги. И все же не удивляйтесь и не унывайте, если библиотека стандартных шаблонов покажется вам слишком запутанной. Просто будьте терпеливы, внимательно изучайте примеры и не позволяйте незнакомому синтаксису заслонять простоту принципов, на которых основана библиотека STL.

В главе рассматривается библиотека стандартных шаблонов, в том числе принципы ее разработки и организации, составляющие части и технология применения. Поскольку библиотека STL очень велика, обсудить все ее особенности в рамках одной главы невозможно. Полному анализу этой библиотеки посвящена часть IV.

Ниже описан один из наиболее важных классов — **`string`**. Этот класс определяет тип данных, позволяющий работать с символьными строками как обычно — с помощью операторов, а не функций. Класс **`string`** является необъемлемой частью библиотеки STL.

Обзор библиотеки STL

Несмотря на то что стандартная библиотека шаблонов велика, а ее синтаксис выглядит устрашающе, с ней довольно легко работать, если правильно понимать, из чего она состоит и как устроена. Следовательно, прежде чем углубиться в примеры, необходимо сделать беглый обзор всей библиотеки.

Библиотека STL покоится на трех китах: *контейнерах*, *алгоритмах* и *итераторах*. Взаимодействие этих элементов обеспечивает стандартные решения очень широкого круга задач программирования.

Контейнеры

Контейнеры — это объекты, хранящие внутри себя другие объекты. Существует несколько видов контейнеров. Например, класс **`vector`** определяет динамический массив, класс **`deque`** создает двустороннюю очередь, а класс **`list`** позволяет работать с линейным списком. Эти контейнеры называются *последовательными* (sequence container), поскольку по терминологии библиотеки STL последовательность — это линейный список. Кроме основных контейнеров в стандартной библиотеке шаблонов существуют *ассоциативные контейнеры* (associative container), позволяющие эффективно извлекать значения по ключу. Например, класс **`map`** обеспечивает доступ к значениям, имеющим уникальные ключи. Таким образом, класс **`map`** хранит пары ключ-значение и позволяет извлекать значение по указанному ключу.

Каждый контейнерный класс определяет набор функций, которые можно применять к контейнеру. Например, класс `list` содержит функции для вставки, удаления и слияния элементов, а класс `stack` предусматривает функции для выталкивания и заталкивания элементов.

Алгоритмы

Алгоритмы применяются к контейнерам. Они позволяют манипулировать содержимым контейнера: инициализировать, сортировать, искать и преобразовывать содержимое контейнера. Многие алгоритмы применяются к целому *диапазону* (range) элементов, находящихся в контейнере. (*Диапазоном* называется последовательность, допускающая произвольный доступ. — *Прим. ред.*)

Итераторы

Итераторы — это объекты, напоминающие указатели. Они позволяют перемещаться по содержимому контейнера так, как указатель перемещается по элементам массива. Существует пять видов итераторов.

Итератор	*Вид доступа*
Итератор произвольного доступа	Хранит и извлекает значения. Обеспечивает произвольный доступ к элементам.
Двунаправленный итератор	Хранит и извлекает значения. Перемещается вперед и назад.
Прямой итератор	Хранит и извлекает значения. Перемещается только вперед.
Итератор ввода	Извлекает, но не хранит элементы. Перемещается только вперед.
Итератор вывода	Хранит, но не извлекает значения. Перемещается только вперед.

Как правило, итераторы, обладающие более широкими возможностями, можно использовать вместо более слабых итераторов. Например, вместо итератора ввода можно применять прямой итератор.

Итераторы действуют как указатели. Их можно увеличивать, уменьшать и разыменовывать, применяя оператор "`*`". Итераторы, т.е. объекты, имеющие тип `iterator`, объявляются в различных контейнерах.

В библиотеке STL существуют также *обратные итераторы* (reverse iterator). Они могут быть либо двунаправленными, либо итераторами произвольного доступа. В обоих случаях они должны иметь возможность перемещаться по контейнеру в обратном направлении. Таким образом, если обратный итератор ссылается на конец последовательности, его увеличение приведет к перемещению на предыдущий элемент.

Упоминая типы итераторов, использованные в описаниях шаблонов, мы будем придерживаться следующей терминологии.

Термин	*Смысл*
`BiIter`	Двунаправленный итератор
`ForIter`	Прямой итератор
`InIter`	Итератор ввода
`OutIter`	Итератор вывода
`RandIter`	Итератор произвольного доступа

Другие элементы библиотеки STL

Кроме контейнеров, алгоритмов и итераторов в библиотеке STL предусмотрены другие стандартные компоненты. Наиболее важными среди них являются рас-

пределители (allocators), предикаты (predicates), функции сравнения (comparison function) и функторы (function objects). (Иногда функторы называют *объектами-функциями. — Прим. ред.*)

Каждый итератор имеет *распределитель*, подобранный именно для него. Распределители управляют выделением памяти для контейнера. Распределитель, предусмотренный по умолчанию, является объектом класса **allocator**. При необходимости можно найти свой собственный распределитель, ориентированный на специфические приложения, но в большинстве случаев можно обойтись распределителем по умолчанию.

Некоторые алгоритмы и контейнеры используют специальный тип функции, называемый *предикатом*. Существуют два вида предикатов: унарные и бинарные. *Унарный предикат* имеет один аргумент, а *бинарный* — два. Эти функции возвращают логические значения **true** или **false**. Однако условия, от которых зависят эти значения, программист может формулировать самостоятельно. В дальнейшем унарные предикаты мы будем относить к типу **UnPred**, а бинарные — к типу **BinPred**. Аргументы бинарных предикатов всегда перечисляются по порядку: *first*, *second*. Аргументами как унарных, так и бинарных предикатов являются объекты, хранящиеся в контейнере.

Некоторые алгоритмы и классы используют специальную разновидность бинарных предикатов, предназначенных для сравнения двух элементов, — функции сравнения. Они возвращают значение **true**, если первый аргумент меньше второго. Функции сравнения имеют тип **Comp**.

Кроме заголовков, характерных для разных шаблонных классов библиотеки STL, в ней используются заголовки **<utility>** и **<functional>**. Например, шаблонный класс **pair**, хранящий пары значений, определен в заголовке **<utility>**.

Шаблоны, определенные в заголовке **<functional>**, позволяют создавать объекты, в которых определена операторная функция **operator()**. Такие объекты называются *функторами* (function objects). Их часто применяют вместо указателей на функции. В заголовке **<functional>** определено несколько встроенных функторов.

plus	**minus**	**multiplies**	**divides**	**modulus**
negate	**equal_to**	**not_equal_to**	**greater**	**greater_equal**
less	**less_equal**	**logical_and**	**logical_or**	**logical_not**

Наиболее широко используется функтор **less**, определяющий отношение между объектами. Функторы можно применять вместо указателей на функции в шаблонных алгоритмах. Использование функторов вместо указателей на функции повышает эффективность кода.

В стандартной библиотеке шаблонов широко применяются еще две разновидности функторов: *редакторы связей* (binders) и *инверторы* (negators). Редакторы связей связывают аргумент с функтором. Инвертор возвращает отрицание предиката.

Осталось дать определение *адаптера*. Проще говоря, адаптер превращает одну сущность в другую. Например, контейнер **queue**, создающий стандартную очередь, является адаптером по отношению к контейнеру **deque**.

Контейнерные классы

Как мы уже говорили, контейнеры представляют собой объекты, в которых хранятся данные. Контейнеры, определенные в стандартной библиотеке шаблонов, перечислены в табл. 24.1.

Таблица 24.1. Контейнеры, определенные в библиотеке STL

Контейнер	*Описание*	*Заголовок*
`bitset`	Набор битов	`<bitset>`
`deque`	Двусторонняя очередь	`<deque>`
`list`	Линейный список	`<list>`
`map`	Хранит пары ключ-значение, в которых каждому ключу соответствует лишь одно значение	`<map>`
`multimap`	Хранит пары ключ-значение, в которых каждому ключу может соответствовать несколько значений	`<map>`
`multiset`	Множество, в которое каждый элемент может входить несколько раз	`<set>`
`priority_queue`	Очередь с приоритетами	`<queue>`
`queue`	Очередь	`<queue>`
`set`	Множество, в которое каждый элемент входит лишь один раз	`<set>`
`stack`	Стек	`<stack>`
`vector`	Динамический массив	`<vector>`

Поскольку имена обобщенных типов, использующихся в качестве параметров шаблонных классов, выбираются произвольно, в контейнерных классах они переопределяются с помощью оператора `typedef`. Это позволяет конкретизировать имена этих типов. Перечислим имена, которые чаще всего переопределяются в помощью оператора `typedef`.

`size_type`	Некая разновидность целочисленного типа
`reference`	Ссылка на элемент
`const_reference`	Константная ссылка на элемент
`iterator`	Итератор
`const_iterator`	Константный итератор
`reverse_iterator`	Обратный итератор.
`const_reverse_iterator`	Константный обратный итератор
`value_type`	Тип значения, хранящегося в контейнере
`allocator_type`	Тип распределителя
`key_type`	Тип ключа
`key_compare`	Тип функции, сравнивающей два ключа
`value_compare`	Тип функции, сравнивающей два значения

Общие принципы функционирования

Несмотря на сложное внутреннее устройство библиотеки STL, использовать ее довольно легко. Во-первых, необходимо выбрать тип контейнера. Каждый из них обладает как преимуществами, так и недостатками. Например, класс `vector` хорош при работе с объектами, обеспечивающими прямой доступ, массивами. Класс `list` позволяет эффективно вставлять и удалять элементы, но снижает быстродействие программы в целом. Класс `map` создает ассоциативный контейнер, однако это сопряжено с дополнительными расходами памяти.

Выбор типа контейнера определяет функции-члены, которые будут использоваться для вставки, модификации и удаления элементов, хранящихся в нем. За исключением класса `bitset`, все контейнеры автоматически увеличиваются по мере добавления элементов и уменьшаются при их удалении.

Существует много способов вставки и удаления элементов контейнера. Например, последовательные (`vector`, `list` и `deque`) и ассоциативные контейнеры (`map`, `multimap`,

set и **multiset**) содержат функцию-член **insert()**, вставляющую элементы в контейнер, и функцию-член **erase()**, удаляющую их из контейнера. Кроме того, последовательные контейнеры содержат функцию-член **push_back()**, добавляющую новый элемент в конец последовательности, и функцию-член **pop_back()**, удаляющую его оттуда. Эти функции используются для вставки и удаления элементов чаще других. Контейнеры **list** и **deque** также содержат функцию-член **push_front()**, добавляющую новый элемент в начало последовательности, и функцию-член **pop_front()**, удаляющую его оттуда.

Как правило, доступ к элементам контейнера обеспечивается с помощью итераторов. Последовательности и ассоциативные контейнеры содержат функции-члены **begin()** и **end()**, возвращающие итераторы, ссылающиеся соответственно на начало и конец контейнера. Эти итераторы очень удобны для доступа к элементам массива. Например, вызвав функцию-член **begin()**, можно получить итератор, установленный на начало массива, а затем увеличивать его, пока его значение не совпадет со значением функции **end()**.

Ассоциативные контейнеры содержат функцию-член **find()**, которая обнаруживает элемент контейнера по заданному ключу. Поскольку ассоциативные контейнеры связывают ключ с его значением, функция **find()** позволяет определить, как расположены элементы контейнера.

Класс **vector** является динамическим массивом, поэтому он поддерживает обычные синтаксические конструкции, применяющиеся для индексации его элементов.

Информация, хранящаяся в контейнере, может обрабатываться несколькими алгоритмами. Эти алгоритмы позволяют не только изменять содержимое массива, но и преобразовывать один тип последовательности в другой.

В следующих разделах мы научимся применять эти общие способы к трем наиболее типичным контейнерам: **vector**, **list** и **map**. Поняв, как работают эти алгоритмы, легко разобраться со всеми остальными.

Векторы

Наиболее универсальным контейнерным классом является класс **vector**, представляющий собой динамический массив, размеры которого могут изменяться по мере необходимости. Как известно, в языке C++ размер массива фиксируется на этапе компиляции. Этот способ определения массивов наиболее эффективен. Однако он связан с серьезными ограничениями, поскольку размер массива в ходе программы изменяться не может. Класс **vector** разрешает эту проблему, выделяя столько динамической памяти, сколько потребуется. Несмотря на то что вектор является динамическим массивом, для доступа к его элементам можно использовать обычный способ индексации.

Шаблонная спецификация класса **vector** выглядит следующим образом.

```
template <class T, class Allocator = allocator<T> > class vector
```

Здесь символ **T** обозначает тип данных, хранящихся в контейнере, а класс **Allocator** определяет механизм распределения памяти. По умолчанию используется стандартный распределитель памяти. Класс **vector** содержит следующие конструкторы.

```
explicit vector(const Allocator &a = Allocator());
explicit vector(size_type num, const T &val = T(),
                const Allocator &a = Allocator());
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
                const Allocator &a = Allocator());
```

Первый конструктор создает пустой вектор. Второй конструктор создает вектор, состоящий из *num* элементов, имеющих значение *val*, которое можно задавать по умолчанию. Третий конструктор создает вектор, содержащий элементы вектора *ob*. Четвертый конструктор создает вектор, состоящий из элементов, лежащих в диапазоне, определенном итераторами *start* и *end*.

Чтобы обеспечить максимальную гибкость и машинонезависимость, любой объект, хранящийся в векторе, должен определять конструктор по умолчанию, а также операторы "**<**" и "**==**". Некоторые компиляторы налагают на такие объекты дополнительные ограничения. (Точная информация об этих ограничениях содержится в технической документации компилятора.) Все встроенные типы по умолчанию удовлетворяют этим условиям.

Хотя шаблонные синтаксические конструкции выглядят довольно сложными, вектор объявляется просто. Рассмотрим несколько примеров.

```
vector<int> iv;           // Создается пустой вектор типа int.
vector<char> cv(5);       // Создается вектор типа char,
                          // состоящий из пяти элементов.
vector<char> cv(5,  'x'); // Инициализируется вектор типа char,
                          // состоящий из пяти элементов.
vector<int> iv2(iv);      // Создается вектор типа int, который
                          // инициализируется другим
                          // целочисленным вектором.
```

В классе **vector** определены следующие операторы сравнения:

==, <, <=, !=, >, >=

Кроме того, в классе **vector** определен оператор "**[]**". Это позволяет обращаться к элементам вектора с помощью стандартной индексации массива.

Функции-члены, определенные в классе **vector**, перечислены в табл. 24.2. (Полное описание шаблонных классов из библиотеки STL приведено в части IV.) К наиболее распространенным функциям — членам класса **vector** относятся функции-члены **size()**, **begin()**, **end()**, **push_back()**, **insert()** и **erase()**. Функция **size()** возвращает текущий размер вектора. Эта функция довольно полезна, поскольку с ее помощью можно определить размер вектора в ходе выполнения программы. Напомним, что векторы могут изменять свои размеры по мере необходимости, поэтому их размер определяется во время выполнения программы, а не на этапе компиляции.

Функция **begin()** возвращает итератор, установленный на начало вектора. Функция **end()** возвращает итератор, установленный на конец вектора.

Функция **push_back()** записывает значение в конец вектора. Если вектор заполнен, при записи его размер увеличивается. С помощью функции **insert()** значение можно записать в середину вектора. Кроме того, вектор можно инициализировать. В любом случае для доступа к элементам вектора можно применять обычные обозначения индексации. Для удаления элемента из вектора предназначена функция **erase()**.

Таблица 24.2. Функции, определенные в классе vector

Функция-член	*Описание*
reference_back(); **const_reference back() const;**	Возвращает ссылку на последний элемент вектора
iterator begin(); **const_iterator begin() const;**	Возвращает итератор, установленный на первый элемент вектора
void clear();	Удаляет из вектора все элементы
bool empty() const;	Возвращает значение **true**, если вектор пуст, в противном случае возвращает значение **false**

Окончание табл. 24.2

Функция-член	Описание
`iterator end();` `const_iterator end() const;`	Возвращает итератор, установленный на первый элемент вектора
`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на элемент, следующий за удаленным
`iterator erase` `(iterator start, iterator end);`	Удаляет элементы из диапазона, заданного итераторами *start* и *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным
`reference front();` `const_reference front() const;`	Возвращает ссылку на первый элемент вектора
`iterator insert` `(iterator i, const T &val);`	Вставляет элемент *val* перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на элемент *val*
`void insert(iterator i,` `size_type num, conet T &val);`	Вставляет *num* копий элемента *val* перед элементом, на который ссылается итератор *i*

`template <class inIter>`
`void insert(iterator i,`
`InIter start, InIter end);`
	Вставляет перед элементом, на который ссылается итератор *i*, последовательность элементов, заданную итераторами *start* и *end*

`reference operator[]`
`(size_type i) const;`
`const_reference operator[]`
`(size_type i);`
	Возвращает ссылку на элемент, заданный итератором *i*

`void pop_back();`	Удаляет последний элемента вектора
`void push_back(const T &val);`	Добавляет элемент *val* в конец вектора
`size_type size() const;`	Возвращает текущее количество элементов вектора

Рассмотрим пример, иллюстрирующий основные операции над векторами.

```
// Демонстрация основных операций над векторами.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // Создаем вектор из 10 элементов.
  int i;

  // Выводим на экран исходный размер вектора v
  cout << "Размер = " << v.size() << endl;

  // Присваиваем элементам вектора определенные значения.
  for(i=0; i<10; i++) v[i] = i + 'a';

  // Выводим на экран содержимое вектора.
  cout << "Текущее содержимое:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  cout << "Расширенный вектор\n";
  /* Записываем в конец вектора новые элементы, при этом
```

```
    его размер увеличивается. */
  for(i=0; i<10; i++) v.push_back(i + 10 + 'a');

  // Выводим на экран текущий размер вектора v.
  cout << "Новый размер = " << v.size() << endl;

  // Выводим на экран содержимое вектора.
  cout << "Текущее содержимое:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // Изменяем содержимое вектора.
  for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
  cout << "Модифицированное содержимое:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Размер = 10
Текущее содержимое:
a b c d e f g h i j

Расширенный вектор
Новый размер = 20
Текущее содержимое:
a b c d e f g h i j k l m n o p q r s t

Модифицированное содержимое:
A B C D E F G H I J K L M N O P Q R S T
```

Рассмотрим эту программу внимательнее. В функции **main()** создается вектор символов **v**, длина которого равна 10. Этот факт подтверждается вызовом функции-члена **size()**. Затем 10 элементов вектора **v** инициализируются символами, начиная с буквы **a** и завершая буквой **j**. Обратите внимание на то, что при этом применяется стандартная индексация. Затем с помощью функции **push_back()** в конец вектора **v** добавляется еще 10 элементов. Для того чтобы записать эти 10 элементов, размер вектора **v** увеличивается. Как свидетельствуют результаты работы программы, после этой операции размер массива становится равным 20. В итоге значения элементов вектора **v** изменяются, причем для доступа к ним используется стандартная индексация.

В этой программе есть еще один интересный момент. Обратите внимание на то, что циклы, отображающие на экране содержимое массива **v**, используют в качестве верхнего предела значение функции **v.size()**. Одно из преимуществ векторов над обычными массивами заключается в том, что текущий размер вектора можно определить в любой момент. Легко убедиться, что это свойство оказывается довольно полезным во многих ситуациях.

Доступ к элементам вектора с помощью итератора

Как известно, массивы и указатели тесно связаны между собой. Доступ к элементам массива осуществляется либо через указатель, либо по индексу. В библиотеке STL наблюдается аналогичная ситуация, только роли массивов и указателей играют векторы и итераторы. Доступ к элементам вектора обеспечивается по индексу либо через итератор. Рассмотрим соответствующий пример.

```
// Доступ к элементам вектора с помощью итератора.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
  vector<char> v(10); // Создаем вектор из 10 элементов.
  vector<char>::iterator p; // Создаем итератор.
  int i;

  // Присваиваем элементам вектора определенные значения.
  p = v.begin();
  i = 0;
  while(p != v.end()) {
    *p = i + 'a';
    p++;
    i++;
  }

  // Выводим содержимое вектора на экран.
  cout << "Исходное содержимое:\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // Изменяем содержимое вектора.
  p = v.begin();
  while(p != v.end()) {
    *p = toupper(*p);
    p++;
  }

  // Выводим содержимое вектора на экран.
  cout << "Модифицированное содержимое\n";
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Исходное содержимое:
a b c d e f g h i j

Модифицированное содержимое:
A B C D E F G H I J
```

Обратите внимание на то, как объявлен итератор **p**. Тип **iterator** определен в контейнерном классе. Следовательно, чтобы получить итератор для элементов конкрет-

ного контейнера, необходимо использовать объявление, аналогичное показанному в примере: просто указывать имя контейнера и спецификатор **iterator**. В данной программе итератор **p** установлен в начало вектора. Для этого используется функция **begin()**. Эта процедура аналогична применению указателей для доступа к элементам массива. Для того чтобы распознать конец вектора, вызывается функция **end()**. Эта функция возвращает итератор, установленный на элемент, следующий за последним элементом вектора. Таким образом, если итератор **p** равен значению **v.end()**, значит, обнаружен конец вектора.

Вставка и удаление элементов вектора

Итак, мы уже знаем, как вставить новые значения в конец вектора. Теперь посмотрим, как их записать в середину. Для этого применяется функция **insert()**. Удалить элементы из вектора можно с помощью функции **erase()**. Рассмотрим программу, иллюстрирующую применение функций **insert()** и **erase()**.

```
// Иллюстрация функций insert и erase.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v(10);
  vector<char> v2;
  char str[] = "<Vector>";
  int i;

  // Инициализируем вектор v.
  for(i=0; i<10; i++) v[i] = i + 'a';

  // Копируем символы из строки str в вектор v2.
  for(i=0; str[i]; i++) v2.push_back(str[i]);

  // Выводим на экран исходное содержимое вектора
  cout << "Исходное содержимое вектора v:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  vector<char>::iterator p = v.begin();
  p += 2; // Устанавливаем итератор на третий элемент.

  // Вставляем 10 символов 'X' в вектор v.
  v.insert(p, 10, 'X');

  // Выводим на экран содержимое вектора после вставки.
  cout << "Размер вектора после вставки символов 'X' = "
       << v.size() << endl;
  cout << "Содержимое вектора v после вставки:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // Удаляем эти элементы.
  p = v.begin();
  p += 2; // Устанавливаем итератор на третий элемент.
  v.erase(p, p+10); // Удаляем следующие 10 элементов.

  // Выводим на экран содержимое вектора после удаления.
```

```
  cout << "Размер вектора v после удаления = "
       << v.size() << endl;
  cout << "Содержимое вектора v после удаления:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << "\n\n";

  // Вставка вектора v2 в вектор v.
  v.insert(p, v2.begin(), v2.end());
  cout << "Размер вектора v после вставки вектора v2 = ";
  cout << v.size() << endl;
  cout << "Содержимое вектора v после вставки:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Исходное содержимое вектора v:
a b c d e f g h i j

Размер вектора после вставки символов 'X' = 20
Содержимое вектора v после вставки:
a b X X X X X X X X X X c d e f g h i j

Размер вектора v после удаления = 10
Содержимое вектора v после удаления:
a b c d e f g h i j

Размер вектора v после вставки вектора v2 = 18
Содержимое вектора v после вставки:
a b < V e c t o r > c d e f g h i j
```

Эта программа демонстрирует применение двух видов функции **insert()**. В первом случае она вставляет в вектор 10 символов **'X'**. Во втором — вставляет в вектор **v** содержимое вектора **v2**. Вторая форма функции **insert()** намного интереснее. Она имеет три аргумента, которые являются итераторами. Первый из них задает позицию, с которой начинается вставка элементов в контейнер. Следующие два аргумента задают начало и конец вставляемой последовательности.

Вектор, содержащий объекты класса

В предыдущих примерах вектор содержал объекты встроенных типов, однако класс **vector** обладает более широкими возможностями. Вектор может состоять из объектов любого типа, включая объекты классов, определенных программистом. Рассмотрим пример, в котором вектор хранит данные о температуре воздуха, измеренной в течение недели. Обратите внимание на то, что класс **DailyTemp** имеет конструктор по умолчанию, а также перегруженные версии операторов "**<**" и "**==**". Помните, что в зависимости от конкретной реализации библиотеки STL перегрузка операторов сравнения может быть необязательной.

```
// Вектор, содержащий объекты класса.
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

class DailyTemp {
  int temp;
```

```
public:
  DailyTemp() { temp = 0; }
  DailyTemp(int x) { temp = x; }

  DailyTemp &operator=(int x) {
    temp = x; return *this;
  }

  double get_temp() { return temp; }
};

bool operator<(DailyTemp a, DailyTemp b)
{
  return a.get_temp() < b.get_temp();
}

bool operator==(DailyTemp a, DailyTemp b)
{
  return a.get_temp() == b.get_temp();
}

int main()
{
  vector<DailyTemp> v;
  int i;

  for(i=0; i<7; i++)
    v.push_back(DailyTemp(60 + rand()%30));

  cout << "Температура по Фаренгейту:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  cout << endl;

  // Преобразование шкалы Фаренгейта в шкалу Цельсия.
  for(i=0; i<v.size(); i++)
    v[i] = (v[i].get_temp()-32) * 5/9 ;

  cout << "Температура в градусах:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";

  return 0;
}
```

Ниже приводится примерный вывод этот программы.

```
Температура по Фаренгейту:
71 77 64 70 89 64 78

Температура по Цельсию:
21 25 17 21 31 17 25
```

Векторам присуща большая мощь, безопасность и гибкость, но, к сожалению, они менее эффективны, чем обычные массивы. Следовательно, во многих ситуациях следует предпочесть обычные массивы. Однако нужно иметь в виду, что в некоторых случаях преимущества класса **vector** перевешивают его недостатки.

Списки

Класс **list** создает двунаправленный линейный список. В отличие от вектора, предоставляющего произвольный доступ к своим элементам, доступ к элементам списка может быть лишь последовательным. Поскольку список является двунаправленным, можно перемещаться от начала к концу и от конца к началу списка.

Шаблонная спецификация класса **list** имеет следующий вид.

```
template <class T, class Allocator = allocator<T> > class list
```

Здесь шаблонный параметр T задает тип данных, хранящихся в списке. Распределитель памяти задается классом **Allocator**, причем по умолчанию используется стандартный механизм распределения памяти. Класс **list** имеет следующие конструкторы.

```
explicit list(const Allocator &a = Allocator());
explicit list(size_type num, const T &val = T(),
              const Allocator &a = Allocator());
list(const list<T,Allocator> &ob);
template <class InIter>list(InIter start, InIter end,
          const Allocator &a = Allocator());
```

Первая версия конструктора создает пустой список. Вторая — список, состоящий из *num* элементов, имеющих значение *val*, причем это значение можно задавать по умолчанию. Третий вариант конструктора создает список, содержаший элемент объекта *ob*. Четвертая версия конструктора формирует список, состоящий из элементов, лежащих в диапазоне, заданном итераторами *start* и *end*.

В классе **list** определены следующие операторы сравнения:

```
==, <, <=, !=, >, >=
```

Наиболее важные функции-члены, определенные в классе **list**, перечислены в табл. 24.3. Как и в классе **vector**, функция **push_back()** записывает значение в конец списка. Для вставки нового элемента в начало списка предназначена функция **push_front()**. С помощью функции **insert()** значение можно записать в середину списка. Два вектора можно объединить, вызвав функцию **splice()**. Кроме того, с помощью функции **merge()** один список можно внедрить в другой.

Таблица 24.3. Функции, определенные в классе list

Функция-член	*Описание*
`reference_back();` `const_reference back() const;`	Возвращает ссылку на последний элемент списка
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент списка
`void clear();`	Удаляет из списка все элементы
`bool empty() const;`	Возвращает значение **true**, если список пуст, в противном случае возвращает значение **false**
`iterator end();` `const_iterator end() const;`	Возвращает итератор, установленный на первый элемент списка
`iterator erase(iterator i);`	Удаляет элемента, на который ссылается итератор *i*. Возвращает итератор, установленный на элемент, следующий за удаленным

Функция-член	Описание
`iterator erase(iterator start, iterator end);`	Удаляет элементы из диапазона, заданного итераторами *start* и *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным
`reference front();` `const_reference front() const;`	Возвращает ссылку на первый элемент списка
`iterator insert (iterator i, const T &val);`	Вставляет элемент *val* перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на элемент *val*
`void insert(iterator i, size_type num, const T &val);`	Вставляет *num* копий элемента *val* перед элементом, на который ссылается итератор *i*
`template <class InIter>` `void insert(iterator i, InIter start, InIter end);`	Вставляет перед элементом, на который ссылается итератор *i*, последовательность элементов, заданную итераторами *start* и *end*

`void merge(list<T,Allocator> &ob);`
`template <class Comp>`
`void merge(list<T,Allocator> &ob, Comp cmpfn);`
	Внедряет упорядоченный список, содержащийся к объекте *ob*, в заданный список. В результате возникает упорядоченный список. После внедрения список, содержащийся в объекте *ob*, становится пустым. Вторая версия функции `merge` получает в качестве параметра функцию, позволяющую сравнивать элементы списка между собой

`void pop_back();`	Удаляет последний элемента списка
`void pop_front();`	Удаляет первый элемент списка
`void push_back(const T &val);`	Добавляет элемент *val* в конец списка
`void push_front(const T &val);`	Добавляет элемент *val* в начало списка
`void remove(const T &val);`	Удаляет из списка элементы, значение которых равно *val*
`void reverse();`	Меняет порядок следования элементов списка на противоположный
`size_type size() const;`	Возвращает текущее количество элементов вектора

`void sort();`
`template <class Comp>`
`void sort(Comp cmpfn);`
	Упорядочивает список. Вторая версия упорядочивает список, используя для сравнения элементов функцию *cmpfn*

`void splice(iterator i, list<T,Allocator> &ob);`	Вставляет в позицию, указанную итератором *i*, содержимое объекта *ob*. После выполнения операции список *ob* становится пустым
`void splice(iterator i, list<T,Allocator> &ob, iterator el);`	Элемент, на который ссылается итератор *el*, удаляется из списка *ob* и вставляется в вызывающий список в позицию, заданную итератором *i*
`void splice(iterator i, list<T,Allocator> &ob, iterator start, iterator end);`	Элементы списка *ob*, расположенные в диапазоне, заданном итераторами *start* и *end*, удаляются из списка *ob* и вставляются в вызывающий список в позицию, заданную итератором *i*

Для достижения гибкости и машинонезависимости любой объект, помещаемый в список, должен иметь конструктор по умолчанию. Кроме того, в нем должен быть определен оператор "<" и, возможно, другие операторы сравнения. Точные требования, предъявляемые к объектам, зависят от конкретного компилятора.

Рассмотрим пример, иллюстрирующий основные операции над списками.

```
// Основные операции над списками.
#include <iostream>
```

```
#include <list>
using namespace std;

int main()
{
  list<int> lst; // Создаем пустой список.
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Размер = " << lst.size() << endl;

  cout << "Содержимое: ";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  // Изменяем содержимое списка.
  p = lst.begin();
  while(p != lst.end()) {
    *p = *p + 100;
    p++;
  }

  cout << "Модифицированное содержимое: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Размер = 10
Содержимое: 0 1 2 3 4 5 6 7 8 9

Модифицированное содержимое: 100 101 102 103 104 105 106 107 108 109
```

Эта программа создает список целых чисел. Сначала формируется пустой список. Затем с помощью функции **push_back()** в него записываются 10 целых чисел, причем каждое следующее число записывается в конец существующего списка. После этого на экран выводятся размер и содержимое самого списка. Для вывода содержимого списка на экран используется итератор.

```
list<int>::iterator p = lst.begin();
while(p != lst.end()) {
  cout << *p << " ";
  p++;
}
```

В этом фрагменте итератор **p** сначала устанавливается на первую позицию списка. Затем при каждом проходе цикла итератор **p** увеличивается на единицу и перемещается на следующий элемент. Выполнение цикла завершается, когда итератор **p** указывает

на конец списка. По существу, этот цикл не отличается от цикла, примененного выше для перемещения по вектору. Такой цикл широко используется в библиотеке STL. Тот факт, что одну и ту же конструкцию можно применять к разным типам контейнеров, свидетельствует о мощи библиотеки STL.

Функция end()

Настало время описать одно довольно неожиданное свойство функции **end()**. Она возвращает указатель не на последний элемент контейнера, а на *следующий* за ним элемент. Таким образом, последнему элементу контейнера соответствует значение **end()-1**. Это позволяет создавать очень эффективные алгоритмы обхода всех элементов контейнера, включая последний элемент. Если значение итератора становится равным значению функции **end()**, значит, все элементы контейнера пройдены. Это свойство всегда следует иметь в виду, поскольку оно довольно неестественно. Рассмотрим следующую программу, которая выводит на экран элементы списка в прямом и обратном порядке.

```
// Функция end().
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // Создаем пустой список.
  int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Содержимое списка в прямом порядке:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "Содержимое списка в обратном порядке:\n";
  p = lst.end();
  while(p != lst.begin()) {
    p--; // Уменьшает указатель.
    cout << *p << " ";
  }

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Содержимое списка в прямом порядке:
0 1 2 3 4 5 6 7 8 9

Содержимое списка в обратном порядке:
9 8 7 6 5 4 3 2 1 0
```

Фрагмент кода, предназначенный для вывода содержимого списка на экран, полностью совпадает с предыдущими примерами. Однако следует обратить особое внимание на фрагмент кода, который выводит содержимое списка в обратном порядке. Сначала итератор **p** с помощью функции **end()** устанавливается на конец списка. Поскольку

функция **end()** возвращает итератор, установленный на объект, расположенный за последним объектом в списке, перед использованием итератор **p** следует уменьшить на единицу. Именно по этой причине итератор **p** в цикле вывода уменьшается перед выполнением оператора **cout**, а не после. Никогда не следует забывать, что функция **end()** возвращает указатель не на последний элемент контейнера, а на *следующий* за ним элемент.

Сравнение функций push_front() и push_back()

Список можно создавать, добавляя элементы либо в его конец, либо в начало. До сих пор мы добавляли элементы только в конец списка и применяли для этого функцию **push_back()**. Для того чтобы добавить элементы в начало списка, следует вызвать функцию **push_start()**. Рассмотрим пример.

```
/* Демонстрация различий между функциями
   push_back() и push_front(). */
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  int i;

  for(i=0; i<10; i++) lst1.push_back(i);
  for(i=0; i<10; i++) lst2.push_front(i);

  list<int>::iterator p;

  cout << "Содержимое списка lst1:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << "\n\n";

  cout << "Содержимое списка lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Эта функция выводит на экран следующие результаты.

```
Содержимое списка lst1:
0 1 2 3 4 5 6 7 8 9

Содержимое списка lst2:
9 8 7 6 5 4 3 2 1 0
```

Поскольку при создании списка **lst2** элементы добавлялись в начало, порядок их следования противоположен порядку следования элементов в списке **lst1**, при создании которого они добавлялись в конец.

Сортировка списка

Список можно упорядочить, вызвав функцию-член **sort()**. Следующая программа создает список, состоящий из случайных целых чисел, а затем сортирует их в порядке возрастания.

```
// Сортировка списка.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<int> lst;
  int i;

  // Создаем список, состоящий из случайных целых чисел.
  for(i=0; i<10; i++)
    lst.push_back(rand());

  cout << "Исходное содержимое:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // Упорядочиваем список.
  lst.sort();

  cout << "Упорядоченное содержимое:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Вот как выглядит примерный вывод этой программы.

```
Исходный список:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Упорядоченный список:
41 6334 11478 15724 18467 19169 24464 26500 269262 29358
```

Вставка одного списка в другой

Упорядоченный список можно вставить в другой список. Результатом этой операции является упорядоченный список, состоящий из элементов исходных списков. Новый список хранится в вызывающем объекте, а второй список становится пустым. Эта операция иллюстрируется следующим примером. Первый список состоит из четных чисел от 0 до 9. Второй список состоит из нечетных чисел. После слияния этих списков образуется последовательность 0 1 2 3 4 5 6 7 8 9.

```
// Слияние двух списков.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back(i);
  for(i=1; i<11; i+=2) lst2.push_back(i);

  cout << "Содержимое списка lst1:\n";
  list<int>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  cout << "Содержимое списка lst2:\n";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // Объединяем два списка.
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "Теперь список lst2 пуст\n";

  cout << "Содержимое списка lst1 после слияния:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Рассмотрим результаты работы этой программы.

```
Содержимое списка lst1:
0 2 4 6 8

Содержимое списка lst1:
1 3 5 7 9

Теперь список lst2 пуст
Содержимое списка lst1 после слияния:
0 1 2 3 4 5 6 7 8 9
```

Этот пример имеет еще одну особенность, связанную с применением функции **empty()**. Если вызывающий контейнер пуст, она возвращает значение **true**. Обратите внимание на то, что функция **merge()** удаляет все элементы из списка, подлежащего вставке. Результаты работы программы это подтверждают.

Список, содержащий объекты класса

Рассмотрим пример, в котором используется список объектов класса **myclass**. Обратите внимание на то, что в этом классе перегружаются операторы "<", ">", "!=" и "==". (При работе с некоторыми компиляторами это делать не обязательно, в то же время другие компиляторы вынуждают перегружать еще и дополнительные операторы.) В библиотеке STL эти операторные функции используются для сравнения объектов, хранящихся в контейнере. Даже если список не упорядочен, иногда приходится сравнивать элементы при поиске, сортировке или слиянии.

```
// Список, содержащий объекты.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass {
  int a, b;
  int sum;
public:
  myclass() { a = b = 0; }
  myclass(int i, int j) {
    a = i;
    b = j;
    sum = a + b;
  }
  int getsum() { return sum; }

  friend bool operator<(const myclass &o1,
                        const myclass &o2);
  friend bool operator>(const myclass &o1,
                        const myclass &o2);
  friend bool operator==(const myclass &o1,
                         const myclass &o2);
  friend bool operator!=(const myclass &o1,
                         const myclass &o2);
};

bool operator<(const myclass &o1, const myclass &o2)
{
  return o1.sum < o2.sum;
}

bool operator>(const myclass &o1, const myclass &o2)
{
  return o1.sum > o2.sum;
}

bool operator==(const myclass &o1, const myclass &o2)
{
  return o1.sum == o2.sum;
}

bool operator!=(const myclass &o1, const myclass &o2)
{
  return o1.sum != o2.sum;
}
```

```
int main()
{
  int i;

  // Создаем первый список.
  list<myclass> lst1;
  for(i=0; i<10; i++) lst1.push_back(myclass(i, i));

  cout << "Первый список: ";
  list<myclass>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // Создаем второй список.
  list<myclass> lst2;
  for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));

  cout << "Второй список: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // Выполняем слияние списков lst1 и lst2.
  lst1.merge(lst2);

  // Выводим на экран объединенный список.
  cout << "Объединенный список: ";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }

  return 0;
}
```

Программа создает два списка, содержащих объекты класса **myclass**, и выводит на экран содержимое каждого списка. Затем она выполняет слияние списков и выводит результат, приведенный ниже.

```
Первый список: 0 2 4 6 8 10 12 14 16 18
Второй список: 0 5 10 15 20 25 30 35 40 45
Объединенный список: 0 0 2 4 5 6 8 10 12 14 15 16 18 20 25 30 35 40 45
```

Ассоциативные контейнеры

Класс **map** создает ассоциативный контейнер, в котором каждому ключу соответствует единственное значение. По существу, ключ представляет собой имя, с помощью которого можно получить требуемое значение. Если в контейнере хранится некое значение, доступ

к нему возможен только через ключ. Таким образом, ассоциативный контейнер фактически хранит пары ключ-значение. Преимущество ассоциативных массивов заключается в доступе к значениям по их ключам. Например, можно создать ассоциативный контейнер, в котором ключом является имя человека, а значением — номер его телефона. В настоящее время ассоциативные контейнеры получают все более широкое применение.

Как указывалось ранее, ассоциативные контейнеры могут содержать лишь уникальные ключи. Дубликаты не допускаются. Если необходимо создать ассоциативный контейнер, в котором можно хранить дубликаты, следует применять класс **multiset**.

Шаблонная спецификация класса **map** имеет следующий вид.

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<pair<const key,T> > class map
```

Здесь класс **Key** определяет тип ключа, шаблонный параметр **T** задает тип данных, хранящихся в ассоциативном массиве, а функция **Comp** позволяет сравнивать два ключа. По умолчанию в качестве функции **Comp** применяется стандартный функтор **less()**. Распределитель памяти задается классом **Allocator**, причем по умолчанию используется стандартный класс **allocator**.

Класс **map** имеет следующие конструкторы.

```
explicit map(const Comp &cmpfn=Comp(),
             const Allocator &a = Allocator());
map(const map<Key,T,Comp,Allocator> &ob);
template <class InIter>list(InIter start, InIter end,
          const Comp &cmpfn = Comp(),
          const Allocator &a = Allocator());
```

Первая версия конструктора создает пустой ассоциативный массив. Вторая — ассоциативный контейнер, содержащий элементы объекта *ob*. Третий вариант конструктора создает ассоциативный массив, состоящий из элементов, лежащих в диапазоне, заданном итераторами *start* и *end*. Функция *cmpfn* определяет порядок следования элементов массива.

Как правило, любой объект, использующийся в качестве ключа, должен определять конструктор по умолчанию, а также оператор "**<**" и другие операторы сравнения. Специфические требования, предъявляемые к ключам, зависят от компилятора.

В классе **map** определены следующие операторы сравнения:

==, <, <=, !=, >, >=

Некоторые функции-члены, определенные в классе **map**, перечислены в табл. 24.4. В этой таблице класс **key_type** представляет собой тип ключа, а класс **value_type** определяет тип **pair<Key,T>**.

Таблица 24.4. Функции, определенные в классе map

Функция-член	*Описание*
iterator begin(); **const_iterator begin() const;**	Возвращает итератор, устаноаленный на первый элемент ассоциативного массива
void clear();	Удаляет из ассоциативного массива все элементы
size_type count **(const key_type &k) const;**	Возвращает текущее количество дубликатов элемента со значением *k* в ассоциативном массиае
bool empty() const;	Возвращает значение **true**, если ассоциативный массив пуст, в противном случае возаращает значение **false**
iterator end(); **const_iterator end() const;**	Возвращает итератор, установленный на первый элемент ассоциативного массива

Функция-член	Описание
`void erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на элемент, следующий за удаленным
`void erase (iterator start, iterator end);`	Удаляет элементы из диапазона, заданного итераторами *start* и *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным
`size_type erase (const key_type &k);`	Удаляет из ассоциативного массива элементы, имеющие значение *k*
`iterator find(const key_type &k);` `const_iterator find (const key_type &k);`	Возвращает итератор, установленный на указанный ключ. Если ключ не найден, возвращается итератор, установленный на конец массива
`iterator insert (iterator i, const value_type &val);`	Вставляет элемент со значением *val* на место или сразу после элемента, на который ссылается итератор *i*. Возвращается итератор, установленный на этот элемент
`template <class inIter> void insert(InIter start, InIter end);`	Вставляет элементы из диапазона, заданного итераторами *start* и *end*
`pair<iterator,bool> insert(const value_type &val);`	Вставляет элемент со значением *val* в вызывающий ассоциативный контейнер. Возвращает итератор, ссылающийся на вставленный элемент. Элемент вставляется, только если его еще не было в ассоциативном массиве. В случае успеха возвращается объект класса `pair<iterator,true>`. В противном случае возвращается объект класса `pair<iterator,false>`
`mapped_type& operator[] (const key_type &i);`	Возвращает ссылку на элемент, заданный итератором *i*. Если элемента в массиве не было, он вставляется туда
`size_type size() const;`	Возвращает текущее количество элементов вектора

Пары ключ-значение хранятся в ассоциативном массиве как объекты типа **pair**. Его шаблонная спецификация имеет следующий вид.

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type;   // Тип ключа
  typedef Vtype second_type;  // Тип значения
  Ktype first;                // Содержит ключ
  Vtype second;               // Содержит значение

// Конструкторы
pair();
pair(const Ktype &k, const Vtype &v);
template<class A, class B> pair(const<A, B> &ob);
}
```

Как подсказывают комментарии, значение, хранящееся в поле **first**, содержит ключ, а поле **second** хранит значение, соответствующее этому ключу.

Пару можно создать, вызвав либо конструкторы класса **pair**, либо функцию **make_pair()**, создающую объекты класса **pair** на основе информации о типе параметров. Функция **make_pair()** является обобщенной. Ее прототип приведен ниже.

```
template <class Ktype, class Vtype>
   pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

Как видим, эта функция возвращает объект класса **pair**, состоящий из пары значений типов *Ktype* и *Vtype*. Преимущество функции **make_pair()** заключается в том,

что типы хранящихся объектов определяются компилятором автоматически, и задавать его явно не требуется.

Следующая программа иллюстрирует основные операции над ассоциативным массивам, в котором хранятся пары значений, определяющие взаимное соответствие между прописными буквами и их ASCII-кодами. Таким образом, ключом является символ, а значением — целое число. Пары ключ-значение имеют следующий вид:

```
A  65
B  66
C  67
```

Когда пользователь вводит ключ (т.е. букву от **A** до **Z**), на экран выводится его ASCII-код.

```
// Демонстрация простого ассоциативного массива.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // Записываем пары в ассоциативный массив.
  for(i=0; i<26; i++) {
    m.insert(pair<char, int>('A'+i, 65+i));
  }

  char ch;
  cout << "Введите ключ: ";
  cin >> ch;

  map<char, int>::iterator p;

  // Найти значение по заданному ключу.
  p = m.find(ch);
  if(p != m.end())
    cout << "ASCII-код ключа равен " << p->second;
  else
    cout << "Ключ не найден.\n";

  return 0;
}
```

Обратите внимание на то, что для создания пары ключ-значение применяется шаблонный класс **pair**. Тип данных, определенный этим классом, должен соответствовать типу ассоциативного массива, в который вставляются пары ключ-значение.

После инициализации ассоциативного массива любое значение можно найти по его ключу. Для этого следует вызвать функцию **find()**, которая возвращает итератор, установленный на требуемый элемент или на конец массива, если нужного элемента в массиве не оказалось. Значение, связанное с найденным ключом, содержится в поле **second** класса **pair**.

В предыдущем примере пара ключ-значение создавалась явным образом с помощью конструктора класса **pair<char, int>**. Наряду с этим можно применять функцию **make_pair()**, создающую пары ключ-значение на основе информации о типе параметров. Например, в предыдущей программе оператор

```
m.insert(make_pair((char)('A'+i), 65+i));
```

также вставлял бы пары ключ-значение в объект **m**. Для того чтобы предотвратить автоматическое преобразование результата сложения **i+'A'** в тип **int**, необходимо выполнить приведение к типу **char**. В противном случае тип определялся бы автоматически.

Ассоциативный массив, содержащий объекты

Как и любой другой контейнер, ассоциативный массив можно использовать для хранения объектов классов, определенных программистом. Например, следующая программа создает простую телефонную книжку. Иначе говоря, она создает ассоциативный массив имен, связанных с номерами. Для этого используются два класса с именами **name** и **number**. Поскольку ассоциативный массив хранит упорядоченный список ключей, в программе определяется оператор "**<**" для сравнения объектов класса **name**. Как правило, оператор "**<**" необходимо определять в любом классе, объекты которого используются в качестве ключей. (Некоторые компиляторы требуют, чтобы в таких классах были определены дополнительные операторы сравнения.)

```
// Применение ассоциативного массива
// для создания телефонной книжки.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name {
  char str[40];
public:
  name() { strcpy(str, ""); }
  name(char *s) { strcpy(str, s); }
  char *get() { return str; }

};

// Определяем оператор < для объектов класса name.
bool operator<(name a, name b)
{
   return strcmp(a.get(), b.get()) < 0;
}

class phoneNum {
  char str[80];
public:
  phoneNum() { strcmp(str, ""); }
  phoneNum(char *s) { strcpy(str, s); }
  char *get() { return str; }
};


int main()
{
  map<name, phoneNum> directory;

  // Заносим имена и номера в ассоциативный массив.
  directory.insert(pair<name, phoneNum>(name("Том"),
                   phoneNum("555-4533")));
  directory.insert(pair<name, phoneNum>(name("Крис"),
                   phoneNum("555-9678")));
```

```
  directory.insert(pair<name, phoneNum>(name("Джон"),
                phoneNum("555-8195")));
  directory.insert(pair<name, phoneNum>(name("Рейчел"),
                   phoneNum("555-0809")));

  // Находим номер телефона по указанному имени.
  char str[80];
  cout << "Введите имя: ";
  cin >> str;

  map<name, phoneNum>::iterator p;

  p = directory.find(name(str));
  if(p != directory.end())
    cout << "Номер телефона: " <<  p->second.get();
  else
    cout << "Такого имени в книжке нет.\n";

  return 0;
}
```

Вот как выглядит примерный результат работы программы.

```
Введите имя: Рейчел
Телефонный номер: 555-0809
```

В данном случае каждый элемент ассоциативного массива представляет собой символьный массив, в котором хранится строка, завершающаяся нулевым байтом. Позднее мы найдем более простой способ решить описанную задачу с помощью стандартного типа **string**.

Алгоритмы

Как правило, содержимое контейнеров обрабатывается алгоритмами. Хотя каждый контейнер определяет свой собственный набор основных операций, более сложные действия описываются в виде алгоритмов. Кроме того, они позволяют одновременно работать с двумя разными типами контейнеров. Для доступа к алгоритмам из библиотеки STL в программу необходимо включить заголовок **<algorithm>**.

В библиотеке STL определено большое количество алгоритмов. Они перечислены в табл. 24.5. Все алгоритмы представляют собой шаблонные функции. Это значит, что их можно применять к любому типу контейнеров. Все шаблонные алгоритмы будут описаны в части 4, а в следующем разделе приведен иллюстративный пример.

Таблица 24.5. Стандартные алгоритмы

Алгоритм	*Предназначение*
adjacent_find	Находит пару соседних элементов, совпадающих между собой, и возвращает итератор, ссылающийся на их первое вхождение
binary_search	Выполняет бинарный поиск в упорядоченной последовательности
copy	Копирует последовательность
copy_backward	Аналогичен алгоритму **copy()**, но копирование начинает с последнего элемента
count	Возвращает количество элементов последовательности

Продолжение табл. 24.5

Алгоритм	Предназначение
`count_if`	Возвращает количество элементов последовательности, удовлетворяющих определенному условию
`equal`	Определяет, совпадают ли элементы двух диапазонов
`equal_range`	Возвращает диапазон, в который можно вставить элементы, не нарушая порядок последовательности
`fill` и `fill_n`	Заполняет диапазон заданным значением
`find`	Находит диапазон, содержащий заданное значение, и возвращает итератор, ссылающийся на его первое вхождение
`find_end`	Находит диапазон подпоследовательности. Возвращает итератор, установленный на конец подпоследовательности, содержащейся в заданном диапазоне
`find_first_of`	Находит первый элемент последовательности, совпадающий с элементом из другой последовательности
`find_if`	Находит первый элемент последовательности, удовлетворяющий определенному условию
`for_each`	Применяет функцию к диапазону элементов
`generate` и `generate_n`	Присваивают элементам диапазона значения, возвращенные функцией-генератором
`includes`	Определяет, содержит ли одна последовательность другую
`inplace_merge`	Объединяет один диапазон с другим. Оба диапазона должны быть упорядочены по возрастанию. Результатом является упорядоченная последовательность
`iter_swap`	Меняет местами два значения, на которые ссылаются аргументы
`lexicographical_compare`	Выполняет лексикографическое сравнение двух последовательностей
`lower_bound`	Определяет первый элемент последовательности, который не меньше, чем заданное значение
`make_heap`	Создает кучу на основе последовательности
`max`	Возвращает наибольшее из двух значений
`max_element`	Возвращает итератор, установленный на максимальный элемент диапазона
`merge`	Объединяет две упорядоченные последовательности, помещая результат в третью последовательность
`min`	Возвращает наименьшее из двух значений
`min_element`	Возвращает итератор, ссылающийся на минимальный элемент последовательности
`mismatch`	Находит первое несовпадение двух последовательностей. Возвращаются итераторы на оба элемента
`next_permutation`	Формирует следующую перестановку элементов последовательности
`nth_element`	Упорядочивает последовательность так, чтобы все ее элементы, меньше заданного элемента `E`, предшествовали ему и остальным элементам
`partial_sort`	Упорядочивает диапазон
`partial_sort_copy`	Упорядочивает диапазон, а затем копирует его в результирующую последовательность
`partition`	Упорядочивает последовательность так, чтобы все ее элементы, удовлетворяющие заданному условию, предшествовали всем остальным
`pop_heap`	Меняет местами первый и последний элементы, а затем перестраивает кучу

Алгоритм	Предназначение
`prev_permutation`	Формирует предыдущую перестановку элементов последовательности
`push_heap`	Заталкивает элемент в конец кучи
`random_shuffle`	Перетасовывает последовательность
`remove`, `remove_if`, `remove_copy` и `remove_copy_if`	Удаляет элементы из указанного диапазона
`replace`, `replace_copy`, `replace_if` и `replace_copy_if`	Заменяет элементы указанного диапазона
`reverse` и `reverse_copy`	Меняет порядок следования элементов диапазона на противоположный
`rotate` и `rotate_copy`	Выполняет левую циклическую перестановку элементов
`search`	Находит подпоследовательность внутри заданной последовательности
`search_n`	Находит *n*-е вхождение элемента в последовательность
`set_difference`	Создает последовательность, содержащую несовпадающие элементы двух последовательностей
`set_intersection`	Создает последовательность, содержащую совпадающие элементы двух последовательностей
`set_symmetric_difference`	Создает последовательность, являющуюся симметричной разностью двух последовательностей
`set_union`	Создает последовательность, являющуюся объединением двух последовательностей
`sort`	Упорядочивает диапазон
`sort_heap`	Упорядочивает кучу внутри указанного диапазона
`stable_partition`	Упорядочивает последовательность так, чтобы все элементы, удовлетворяющие определенному условию, предшествовали всем остальным. Разбиение последовательности фиксируется. Это значит, что при упорядочении сохраняется взаимное расположение двух последовательностей
`stable_sort`	Упорядочивает диапазон. Порядок фиксируется. Это значит, что равные элементы остаются на своих местах
`swap`	Меняет местами два элемента
`swap_ranges`	Меняет местами элементы заданного диапазона
`transform`	Применяет функцию к элементам заданного диапазона, сохраняя результат в новой последовательности
`unique` и `unique_copy`	Удаляет дубликаты из заданного диапазона.
`upper_bound`	находит первый элемент последовательности, не превышающий заданное значение

Подсчет

Одной из главных операций, применяющихся к последовательностям, является подсчет их элементов. Для этого используются функции `count()` или `count_if()` Их общий вид приведен ниже.

```
template <class InIter, class T>
  prtdiff_t count(InIter start, InIter end, const T &val);
```

```
template <class InIter, class UnPred>
   prtdiff_t count(InIter start, InIter end, UnPred pfn);
```

Тип **prtdiff_t** является разновидностью типа **int**. Алгоритм **count()** возвращает количество элементов последовательности, имеющих значение *val*, начиная с позиции, заданной итератором *start*, и заканчивая позицией, заданной итератором *end*. Алгоритм **count_if()** возвращает количество элементов последовательности, удовлетворяющих предикату *pfn*, начиная с позиции, заданной итератором *start*, и заканчивая позицией, заданной итератором *end*.

Следующая программа демонстрирует применение алгоритма **count()**.

```
// Демонстрация алгоритма count().
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
  vector<bool> v;
  int i;

  for(i=0; i < 10; i++) {
   if(rand() % 2) v.push_back(true);
   else v.push_back(false);
  }

  cout << "Последовательность:\n";
  for(i=0; i<v.size(); i++)
    cout << boolalpha << v[i] << " ";
  cout << endl;

  i = count(v.begin(), v.end(), true);
  cout << "Предикату соответствуют " << i << " элемента.\n";

  return 0;
}
```

Эта программа выводит на экран следующие результаты.

```
Последовательность:
true true false false true false false false false false
Предикату соответствуют 3 элемента.
```

Сначала программа создает вектор, содержащий случайно сгенерированные значения **true** и **false**. Затем для подсчета значений **true** применяется функция **count()**.

Следующая программа иллюстрирует работу функции **count_if()**. Она создает вектор, содержащий числа от 1 до 19. Затем она подсчитывает элементы, кратные 3. Для этого создается унарный предикат **dividesBy3()**, возвращающий значение **true**, если его аргумент кратен 3.

```
// Демонстрация алгоритма count_if().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
/* Унарный предикат, определяющий, делится ли число на 3. */
bool dividesBy3(int i)
{
  if((i%3) == 0) return true;

  return false;
}

int main()
{
  vector<int> v;
  int i;

  for(i=1; i < 20; i++) v.push_back(i);

  cout << "Последовательность:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl;

  i = count_if(v.begin(), v.end(), dividesBy3);
  cout << "Количество чисел, кратных 3, =" << i << ".\n";

  return 0;
}
```

Результаты работы программы приведены ниже.

```
Последовательность:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Количество чисел, кратных 3, = 6.
```

Обратите внимание на то, как закодирован унарный предикат **divideBy3()**. Аргументами всех унарных предикатов являются объекты, хранящиеся в соответствующем контейнере. Предикат должен возвращать значение **true** или **false**.

Удаление и замена элементов

Иногда необходимо создать новую последовательность, содержащую лишь часть элементов исходной последовательности. Для этого можно применять шаблон алгоритма **remove_copy()**. Его спецификация приведена ниже.

```
template <class InIter, class OutIter, class T>
   OutIter remove_copy(InIter start, InIter end,
                       OutIter result, const T &val);
```

Алгоритм **remove_copy()** копирует элементы из заданного диапазона, удаляя из него элементы, имеющие значение *val*. Он помещает результат в последовательность, на которую ссылается итератор *result*, и возвращает итератор, установленный на ее конец. Размер результирующего контейнера должен быть достаточно большим, чтобы в нем поместился результат.

Чтобы в процессе копирования заменить один элемент последовательности другим, следует применять алгоритм **replace_copy()**. Его спецификация приведена ниже.

```
template <class InIter, class OutIter, class T>
  OutIter remove_copy(InIter start, InIter end,
                      OutIter result, const T &old,  const T &new);
```

Алгоритм **replace_copy()** копирует элементы из заданного диапазона, заменяя элемент *old* элементом *new*. Он помещает результат в последовательность, на которую ссылается итератор *result*, и возвращает итератор, установленный на ее конец. Размер результирующего контейнера должен быть достаточно большим, чтобы в нем поместился результат.

Алгоритмы **remove_copy()** и **replace_copy()** иллюстрируются следующей программой. Она создает последовательность символов, а затем удаляет из нее все пробелы, заменяя их двоеточиями.

```
// Демонстрация алгоритмов remove_copy и replace_copy.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  char str[] = "Шаблон STL обладает большой мощью.";
  vector<char> v, v2(30);
  int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  // **** Демонстрация алгоритма remove_copy ****
  cout << "Исходная последовательность:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // Удаляем все пробелы.
  remove_copy(v.begin(), v.end(), v2.begin(), ' ');

  cout << "Результат после удаления пробелов:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  // **** Демонстрация алгоритма replace_copy ****
  cout << "Исходная последовательность:\n";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  // Заменяем пробелы двоеточиями
  replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');

  cout << "Результат после замены пробелов двоеточиями:\n";
  for(i=0; i<v2.size(); i++) cout << v2[i];
  cout << endl << endl;

  return 0;
}
```

Результат работы этой программы приведен ниже.

```
Исходная последовательность:
Библиотека STL обладает большой мощью.
Результат после удаления пробелов:
БиблиотекаSTLобладаетбольшоймощью.

Исходная последовательность:
Библиотека STL обладает большой мощью.
Результат после удаления пробелов:
Библиотека:STL:обладает:большой:мощью.
```

Изменение порядка следования элементов последовательности

Для изменения порядка следования элементов последовательности на обратный применяется алгоритм **reverse()**. Его спецификация имеет следующий вид.

```
template <class BiIter> void reverse(BiIter start, BiIter end);
```

Алгоритм **reverse()** изменяет порядок следования элементов в диапазоне между итераторами *start* и *end* на обратный.

Проиллюстрируем алгоритм **reverse()** следующей программой.

```
// Демонстрация алгоритма reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<int> v;
  int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Исходная последовательность: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Обратная последовательность: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

Результаты ее работы выглядят так.

```
Исходная последовательность: 0 1 2 3 4 5 6 7 8 9
Обратная последовательность: 9 8 7 6 5 4 3 2 1 0
```

Преобразование последовательности

Один из самых интересных алгоритмов — **transform()** — модифицирует каждый элемент заданного диапазона с помощью функции, заданной программистом. Алгоритм **transform()** имеет две разновидности.

```
template <class InIter, class OutIter, class Func,
  OutIter transform(InIter start, inIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2,
          class OutIter, class Func,
  OutIter transform(InIter1 start1, inIter1 end1,
                    InIter2 start2, OutIter result,
                    Func binaryfunc);
```

Этот алгоритм применяет функцию ко всем элементам заданного диапазона и сохраняет результат в объекте *result*. В первом варианте диапазон определяется итераторами *start* и *end*, функция задается параметром *unaryfunc*. Она получает значения элементов

своего параметра и возвращает преобразованную последовательность. Во втором варианте преобразование последовательности выполняется функтором *binaryfunc*. Первым параметром этой функции является значение элемента из этой последовательности, а вторым — элемент из второй последовательности. Обе версии функции **transform()** возвращают итератор, ссылающийся на конец результирующей последовательности.

Следующая программа использует для преобразования последовательности простую функцию **reciprocal()**, которая заменяет числа, входящие в список, обратными величинами. Обратите внимание на то, что результирующая последовательность записывается в исходный список.

```
// Пример преобразования последовательности.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// Простая функция, задающая преобразование.
double reciprocal(double i) {
  return 1.0/i; // Возвращает обратное число.
return reciprocal
}

int main()
{
  list<double> vals;
  int i;

  // Занести число в список.
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Исходное содержимое списка vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // Преобразование списка vals.
  p = transform(vals.begin(), vals.end(),
                vals.begin(), reciprocal);

  cout << "Преобразованное содержимое списка vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Результат работы этой программы показан ниже.

```
Исходное содержимое списка vals:
1 2 3 4 5 6 7 8 9
Преобразованное содержимое списка vals:
1 0.5 0.33333 0.25 0.2 0.166667 0.142857 0.125 0.111111
```

Применение функторов

Как указывалось в начале главы, библиотека STL интенсивно использует функторы. Напомним, что функтор — это класс, в котором определена операторная функция **operator()**. Библиотека STL содержит большое количество встроенных функторов, например, **less**, **minus** и др. Она позволяет также определять свои собственные функторы. Честно говоря, в задачу нашей книги не входит полное описание процесса создания и применения функторов. К счастью, как показывают предыдущие примеры, библиотеку STL можно применять, не прибегая к помощи функторов. Однако, поскольку функторы являются основным компонентом библиотеки STL, важно понимать, как они работают.

Унарные и бинарные функторы

Подобно унарным и бинарным предикатам, существуют унарные и бинарные функторы. Унарный функтор имеет один аргумент, а бинарный — два. Унарный и бинарный функторы не могут заменять друг друга. Например, если алгоритм использует бинарный функтор, ему нужно передавать именно бинарный, а не унарный функтор.

Применение встроенных функторов

В библиотеке STL предусмотрен широкий выбор встроенных функторов. Перечислим бинарные функторы.

plus	**minus**	**multplies**	**divides**	**modulus**
equal_to	**not_equal_to**	**greater**	**greater_equal**	**less**
less_equal	**logical_and**	**logical_or**		

К унарным относятся следующие функторы.

logical_not	**negate**

Функторы выполняют операции, определенные их именами. Единственное имя, которое не является самоочевидным, — название функтора **negate()**, изменяющего знак своего аргумента.

Встроенные функторы являются шаблонными классами, в которых перегружен оператор **()**, возвращающий результат выбранной операции. Например, чтобы применить бинарный функтор **plus()** к данным типа **float**, следует использовать следующую синтаксическую конструкцию.

```
plus<float>()
```

Для вызова встроенных функторов в программу необходимо включить заголовок **<functional>**.

Рассмотрим простой пример. Следующая программа использует алгоритм **transform()**, описанный в предыдущем разделе, и функтор **negate()** для изменения знака значений, перечисленных в списке.

```
// Применение унарного функтора.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  int i;
```

```
  // Заносим значения в список.
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "Исходное содержимое списка vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // Применение функтора negate.
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                negate<double>()); // Вызываем функтор.

  cout << "Измененное содержимое списка vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Программа выводит на экран следующие результаты.

```
Исходное содержимое списка vals:
1 2 3 4 5 6 7 8 9 0
Измененное содержимое списка vals:
-1 -2 -3 -4 -5 -6 -7 -8 -9 -0
```

Обратите внимание на то, как в этой программе вызывается функтор **negate()**. Поскольку список **vals** хранит числа типа **double**, функтор **negate()** вызывается с помощью конструкции **negate<double>()**. Алгоритм **transform()** автоматически применяет функтор **negate()** к каждому элементу последовательности. Таким образом, единственный параметр функтора **negate()** является элементом последовательности.

Следующая программа демонстрирует применение бинарного функтора **divides()**. Она создает два списка действительных чисел и делит значения одного списка на значения другого. Для этого используется бинарная форма алгоритма **transform()**.

```
// Применение бинарного функтора.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<double> vals;
  list<double> divisors;
  int i;

  // Заносим значения в список.
  for(i=10; i<100; i+=10) vals.push_back((double)i);
  for(i=1; i<10; i++) divisors.push_back(3.0);
```

```
  cout << "исходное значение списка vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  cout << endl;

  // Преобразование списка vals.
  p = transform(vals.begin(), vals.end(),
                divisors.begin(), vals.begin(),
                divides<double>()); // Вызываем функтор.

  cout << "Измененное содержимое списка vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Программа выводит на экран следующие результаты.

```
Исходное содержимое списка vals:
10 20 30 40 50 60 70 80 90
Измененное содержимое списка vals:
3.33333 6.66667 10 13.3333 16.6667 20 23.3333 26.6667 30
```

В данном случае бинарный функтор **divides()** делит элементы первой последовательности на соответствующие элементы второй последовательности. Таким образом, функтор **divides()** получает свои аргументы в следующем порядке.

```
divides(first, second);
```

Этот порядок можно обобщить. Независимо от применяемого бинарного функтора, его аргументы всегда перечисляются в указанном порядке *first, second*.

Создание функтора

Кроме встроенных функторов, можно создавать и применять свои собственные функторы. Для этого достаточно создать класс, в котором перегружается операторная функция **operator()**. Однако, чтобы достичь большей гибкости, для создания функторов следует использовать один из следующих базовых классов, определенных в классе STL.

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};

template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

Эти шаблонные классы конкретизируют имена обобщенных типов данных, используемых функтором. Они весьма удобны, и практически всегда применяются для создания функторов.

Следующая программа демонстрирует обобщенный функтор. Она преобразует функцию **reciprocal()**, использованную при описании алгоритма **transform()**, в функтор.

```
// Создание функтора reciprocal.
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

// Простой функтор.
class reciprocal: unary_function<double, double> {
public:
  result_type operator()(argument_type i)
  {
    return (result_type) 1.0/i; // Возвращаем обратное число.
  }
};

int main()
{
  list<double> vals;
  int i;

  // Заносим значения в список.
  for(i=1; i<10; i++) vals.push_back((double)i);

  cout << "исходное содержимое списка vals:\n";
  list<double>::iterator p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // Применение функтора reciprocal.
  p = transform(vals.begin(), vals.end(),
                vals.begin(),
                reciprocal()); // Вызов функтора.

  cout << "Измененное содержимое списка vals:\n";
  p = vals.begin();
  while(p != vals.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Обратите внимание на два важных аспекта, связанных с функтором **reciprocal()**. Во-первых, он наследует свойства базового класса **unary_function**, а также предоставляет доступ к типам **argument_type** и **result_type**. Во-вторых, этот фактор определяет операторную функцию **operator()**, возвращающую обратное значение аргумента. Как правило, чтобы создать функтор, достаточно создать класс, производный от одного из базовых классов, указанных выше, и перегрузить оператор **()**. Это действительно просто.

Применение редакторов связей

При вызове бинарного функтора один из его аргументов можно связать с конкретным значением. Во многих ситуациях это свойство оказывается довольно полезным. Допустим, что из последовательности требуется удалить все элементы, превышающие определенное значение, например, число 8. Для этого необходимо иметь возможность связывать число 8 с правым операндом функтора **greater()**. Иначе говоря, желательно, чтобы функтор **greater()** мог выполнять сравнение

```
val > 8
```

для каждого элемента последовательности. В библиотеке STL существует механизм, позволяющий это делать. Он называется *редактором связей* (binder).

Существуют два редактора связей — **bind2nd()** и **bind1st()**, имеющие следующий вид.

```
bind1st(binfunc_obj, value);
bind2st(binfunc_obj, value);
```

Здесь параметр *binfunc_obj* является бинарным функтором. Редактор связей **bind1st()** возвращает унарный функтор, у которого левый операнд функтора *binfunc_obj* связан со значением *value*. Редактор связей **bind2st()** возвращает унарный функтор, у которого со значением *value* связан правый операнд функтора *binfunc_obj*. Этот редактор связей используется чаще. В любом случае результатом работы редактора связей является унарный функтор, связанный с определенным значением.

Продемонстрируем работу редактора связей на примере алгоритма **remove_it()**. Он удаляет элементы из последовательности, анализируя результат предиката. Его прототип имеет следующий вид.

```
template <class ForIter, class UnPred>
   ForIter remove_it(ForIter start, ForIter end, UnPred func);
```

Алгоритм удаляет элементы из диапазона, определенного итераторами *start* и *end*, если унарный предикат, заданный параметром *func*, является истинным. Алгоритм возвращает указатель на конец новой последовательности, образованной вследствие удаления элементов.

Следующая программа удаляет из последовательности все элементы, превышающие число 8. Поскольку предикат, необходимый редактору связей **remove_it()**, является унарным, мы не можем просто вызвать бинарный функтор **greater()**. Вместо этого следует связать число 8 со вторым аргументом функтора **greater()**, используя редактор связей **bind2nd()**.

```
// Демонстрация редактора связей bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<int> lst;
  list<int>::iterator p, endp;

  int i;

  for(i=1; i < 20; i++) lst.push_back(i);
```

```
  cout << "Исходная последовательность:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  endp = remove_if(lst.begin(), lst.end(),
                   bind2nd(greater<int>(), 8));

  cout << "Результирующая последовательность:\n";
  p = lst.begin();
  while(p != endp) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Результат работы этой программы приведен ниже.

```
Исходная последовательность:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Результирующая последовательность:
1 2 3 4 5 6 7 8
```

Поэкспериментируйте с этой программой, применяя разные функторы и редакторы связей. Вы убедитесь, насколько редакторы связей увеличивают мощь библиотеки STL.

И последнее: в библиотеке есть объект, тесно связанный с редактором связей. Он называется *инвертором* (negator) и возвращает отрицание (т.е. дополнение) предиката. Его общий вид приведен ниже.

```
not1(unary_predicate);
not2(binary_predicate);
```

Например, если подставить строку

```
endp = remove_if(lst.begin(), lst.end(),
                 not1(bind2nd(greater<int>(), 8)));
```

в предыдущую программу, она удалит из последовательности **lst** все элементы, не превышающие число 8.

Класс string

Как известно, язык C++ не предусматривает встроенного типа для строк. Для работы с ними существует две возможности. Во-первых, можно использовать традиционный массив символов, завершающийся нулевым байтом, с которым мы уже хорошо знакомы. Иногда этот массив называют *С-строкой*. Во-вторых, можно создать объект класса **string**. Рассмотрим этот способ подробнее.

Фактически класс **string** является специализацией более общего шаблонного класса **basic_string**. На самом деле класс **basic_string** имеет две специализации: класс **string**, поддерживающий 8-битовые строки, и класс **wstring**, пред-

назначенный для работы со строками расширенных символов. Поскольку 8-битовые строки до сих пор широко используются в программировании, мы рассмотрим лишь класс **string**.

Прежде чем перейти к изучению класса **string**, следует разобраться, почему его включили в библиотеку языка C++. Стандартные классы обычно не являются частью языка C++. Каждое новшество языка сопровождается длительными и напряженными дебатами. Поскольку язык C++ уже содержал средства для работы с массивами символов, завершающихся нулевым байтом, класс **string**, на первый взгляд, выглядит исключением из этого правила. Однако это неверно, поскольку к строкам, завершающимся нулевым байтом, нельзя применить ни один стандартный оператор языка C++. Такие строки не могут быть частью обычных выражений. Рассмотрим, например, следующий фрагмент.

```
char s1[80], s2[80], s3[80];


s1 = "Альфа"; // Нельзя!
s2 = "Бета";  // Нельзя!
s3 = s1 + s2; // Ошибка, и так нельзя!
```

Как следует из комментариев, в языке C++ невозможно ни присвоить массиву символов новое значение, используя оператор присваивания (за исключением инициализации), ни применить оператор "**+**" для конкатенации двух строк.

Эти операции следует записать, используя библиотечные функции.

```
strcpy(s1, "Alpha");
strcpy(s2, "Beta");
strcpy(s3, s1);
strcat(s3, s2);
```

Поскольку с формальной точки зрения массив символов, завершающийся нулевым байтом, не является типом, к нему нельзя применять операторы языка C++. Это усложняет даже самые простые операции со строками. Именно это ограничение послужило стимулом для разработки стандартного класса **string**. Следует помнить, что, определяя класс в языке C++, вы формируете новый тип данных, который полностью интегрируется в среду языка. Разумеется, это означает, что в новом классе обычные операторы можно перегрузить. Следовательно, стандартный класс **string** позволяет работать со строками, как с любым другим типом данных, а именно: применяя к ним операторы.

Однако существует и другая причина, обосновывающая существование стандартного класса **string**, — безопасность. В руках неопытного или беспечного программиста последний элемент массива, содержащий нулевой байт, совершенно беззащитен. Рассмотрим, например, стандартную функцию **strcpy()**, предназначенную для копирования строк. Эта функция не предусматривает никакой проверки границ результирующего массива. Если исходный массив длиннее результирующего, может возникнуть непоправимая ошибка, приводящая к краху операционной системы.

Итак, существуют три причины, по которым стандартный класс **string** был включен в язык C++: логичность (теперь строка является типом данных), удобство (к строкам можно применять стандартные операторы языка C++) и безопасность (невозможно выйти за пределы массива). Следует заметить, что нет никаких причин отказываться от традиционных массивов символов, завершающихся нулевым байтом. Они по-прежнему являются наиболее эффективным способом реализации строк. Однако, если быстродействие программы не является принципиально важным фактором, новый класс **string** предоставляет безопасный и чрезвычайно удобный способ работы со строками.

Хотя традиционно класс **string** относят к стандартной библиотеке языка C++, а не к библиотеке STL, на самом деле он является разновидностью контейнерных классов. Это значит, что он поддерживает все алгоритмы, описанные в предыдущем разделе. Кроме этого, класс **string** обладает дополнительными возможностями. Для работы с классом **string** необходимо включить в программу заголовок **<string>**.

Класс **string** очень велик. Он содержит большое количество конструкторов и функций-членов. Кроме того, многие функции-члены имеют перегруженные версии. По этой причине класс **string** невозможно описать в одной главе. Вместо этого мы ограничимся наиболее важными и часто применяемыми свойствами. Общее представление о принципах работы класса **string** позволит читателям легко разобраться со всем остальным.

Класс **string** содержит несколько конструкторов. Прототипы трех наиболее важных конструкторов приведены ниже.

```
string();
string(const char *str);
string(const string &str);
```

Первый конструктор создает пустой объект класса **string**. Второй конструктор создает объект класса **string** из строки, завершающейся нулевым байтом, на которую ссылается указатель *str*. Эта версия конструктора преобразует C-строку в объект класса **string**. Третий конструктор создает объект класса **string** на основе другого объекта этого класса.

Для объектов класса **string** определено большое количество операторов.

Оператор	*Значение*
=	Присвоение
+	Конкатенация
+=	Присвоение и конкатенация
==	Равенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
[]	Индексация
<<	Вывод
>>	Ввод

Эти операторы позволяют использовать объекты класса **string** в обычных выражениях и избежать применения функций наподобие **strcpy()** или **strcat()**. Как правило, в одном и том же выражении объекты класса **string** можно смешивать с обычными C-строками. Например, объекту класса **string** можно присвоить строку, завершающуюся нулевым байтом.

Для конкатенации двух объектов класса **string**, а также объекта класса **string** и обычной C-строки можно применять оператор "**+**". Иначе говоря, допускаются следующие варианты.

```
string+string
string+C-строка
C-строка+string
```

Кроме того, с помощью оператора "+" к концу строки можно приписать еще один символ.

В классе **string** определена константа **npos**, равная -1. Эта константа задает максимально возможную длину строки.

Класс **string** позволяет чрезвычайно легко обрабатывать строки. Например, объектам класса **string** можно присваивать строки, заключенные в двойные кавычки, а также сравнивать объекты между собой, пользуясь обычными операторами сравнения. Эти операции иллюстрируются следующей программой.

```
// Короткий пример, демонстрирующий класс string.
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Альфа");
  string str2("Бета");
  string str3("Омега");
  string str4;

  // Присваиваем одну строку другой.
  str4 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // Конкатенируем строки.
  str4 = str1 + str2;
  cout << str4 << "\n";

  // Конкатенируем объект класса string с С-строкой.
  str4 = str1 + " и " + str3;
  cout << str4 << "\n";

  // Сравниваем строки.
  if(str3 > str1) cout << "str3 > str1\n";
  if(str3 == str1+str2)
    cout << "str3 == str1+str2\n";

  /* Объекту класса string можно присвоить обычную строку. */
  str1 = "Это обычная строка, завершающаяся нулем.\n";
  cout << str1;

  // Создает объект класса string, используя
  // другой объект этого класса.
  string str5(str1);
  cout << str5;

  // Ввод строки.
  cout << "Введите строку: ";
  cin >> str5;
  cout << str5;

  return 0;
}
```

Результаты работы этой программы приведены ниже

```
Альфа
Бета
```

```
Омега
Альфа и Омега
str3 > str1
Это обычная строка, завершающаяся нулем.
Это обычная строка, завершающаяся нулем.
Введите строку: STL
STL
```

Обратите внимание на то, как легко теперь работать со строками. Например, с помощью оператора "+" их можно конкатенировать, а оператор ">" позволяет сравнивать строки между собой. Для того чтобы выполнить эти операции в стиле языка С, пришлось бы вызывать функции **strcat()** и **strcmp()**. Поскольку объекты класса **string** и обычные строки можно смешивать в одном и том же выражении, программы становятся намного проще.

Следует отметить еще одну особенность предыдущей программы: размер строки нигде не указывался явно. Объекты класса **string** автоматически вычисляют свой размер, необходимый для хранения соответствующей строки. Таким образом, при присвоении или конкатенации строк результирующая строка автоматически увеличивается на требуемую величину. Это динамическое свойство объектов класса **string** является их несомненным преимуществом по сравнению с обычными строками, в которых *возможен* выход за пределы допустимого диапазона.

Некоторые функции — члены класса string

Хотя основные операции над строками можно выполнить с помощью основных операторов, более сложные действия над строками осуществляются функциями — членами класса **string**. Поскольку класс **string** содержит слишком большое количество функций-членов, мы рассмотрим лишь некоторые из них.

Основные манипуляторы

Чтобы присвоить одну строку другой, необходимо применять функцию **assign()**. Она имеет два варианта.

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const string *str, size_type num);
```

Первая функция присваивает вызывающему объекту *num* символов из строки *strob*, начиная с индекса, заданного параметром *start*. Вторая функция присваивает вызывающему объекту *num* символов из С-строки *str*, начиная с индекса, заданного параметром *start*. В любом случае возвращается ссылка на вызывающий объект. Разумеется, для присвоения полных строк было бы намного легче применять оператор "=". Функцию **assign** следует использовать лишь для присвоения частичной строки.

С помощью функции-члена **append()** можно добавить одной строке часть другой строки. Эта функция имеет две формы.

```
string &append(const string &strob, size_type start, size_type num);
string &append(const string *str, size_type num);
```

Первая функция добавляет к вызывающему объекту *num* символов из строки *strob*, начиная с позиции, заданной параметром *start*. Вторая функция приписывает к вызывающему объекту *num* символов из С-строки *str*, начиная с индекса, заданного параметром *start*. В любом случае возвращается ссылка на вызывающий объект. Разумеется, для конкатенации двух полных строк было бы намного легче применить оператор "+". Функцию **append** следует использовать лишь для приписывания частичной строки.

Функции **insert()** и **replace()** предназначены для вставки и замены символов. Их прототипы перечислены ниже.

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type intStart, size_type num);
string &replace(size_type start, const string &strob);
string &replace(size_type start, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

Первая форма функции **insert()** вставляет строку *strob* в вызывающий объект класса **string**, начиная с индекса, заданного параметром *start*. Второй вариант функции **insert()** вставляет *num* символов строки *strob*, начиная с позиции, заданной параметром *insStart*, в вызывающий объект класса **string**, начиная с индекса, заданного параметром *start*.

Первая форма функции **replace()** заменяет *num* символов вызывающего объекта класса **string**, начиная с позиции, заданной параметром *start*, строкой *strob*. Второй вариант функции **replace()** заменяет *orgNum* символов объекта класса **string**, начиная с позиции, заданной параметром *start*, *replaceNum* символами строки *strob*, начиная с индекса, заданного параметром *replaceStart*. В обоих случаях возвращается ссылка на вызывающий объект.

С помощью функции **erase()** можно удалить символы из строки. Один из ее вариантов выглядит следующим образом.

```
string &erase(size_type start = 0, size_type num = npos);
```

Эта функция удаляет *num* символов из вызывающего объекта, начиная с позиции, заданной параметром *start*.

Функции **insert()**, **erase()** и **replace()** иллюстрируются следующей программой.

```
// Демонстрация функций insert(), erase() и replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("String handling C++style.");
  string str2("Power STL.");

  cout << "Начальные строки:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // Демонстрация функции insert().
  cout << "Вставляем строку str2 в строку str1:\n";
  str1.insert(6, str2);
  cout << str1 << "\n\n";

  // Демонстрация функции erase().
  cout << "Удаляем 9 символов из строки str1:\n";
  str1.erase(6, 9);
  cout << str1 <<"\n\n";

  // Демонстрация функции replace.
```

```
  cout << "Заменяем 8 символов строки str1 строкой str2:\n";
  str1.replace(7, 8, str2);
  cout << str1 << endl;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Исходные строки:
str1: String handling C++ style.
str2: Power STL

Вставляем строку str2 в строку str1:
StringSTL Power handling C++ style.

Удаляем 9 символов из строки str1:
String handling C++ style.

Заменяем 8 символов в строке str1 строкой str2:
String STL Power handling C++ style.
```

Поиск символа в строке

Класс **string** содержит несколько функций-членов, предназначенных для поиска подстрок, в частности функции **find()** и **rfind()**. Их прототипы приведены ниже.

```
size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;
```

Начиная с позиции, заданной параметром *start*, функция **find()** выполняет поиск первого вхождения строки, содержащейся в объекте *strob*. Если подстрока найдена, функция **find()** возвращает индекс ее первого символа в вызывающем объекте. Если подстрока не найдена, возвращается константа **npos**. Функция **rfind()** является противоположностью функции **find()**. Начиная с позиции, заданной параметром *start*, функция **rfind()** выполняет поиск последнего вхождения строки, содержащейся в объекте *strob*. Если подстрока найдена, функция **rfind()** возвращает индекс ее последнего вхождения в вызывающий объект. Если подстрока не найдена, возвращается константа **npos**.

Рассмотрим короткий пример, в котором используются функции **find()** и **rfind()**.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  int i;
  string s1 =
    "Чистые руки, горячее сердце, холодная голова";
  string s2;

  i = s1.find("Чистые");
  if(i!=string::npos) {
    cout << "Найдено соответствие в позиции " << i << endl;
```

```
    cout << "оставшаяся часть строки:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("горячее");
  if(i!=string::npos) {
    cout << "Найдено соответствие в позиции " << i << endl;
    cout << "Оставшаяся часть строки:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  i = s1.find("холодная");
  if(i!=string::npos) {
    cout << "Найдено соответствие в позиции " << i << endl;
    cout << "Оставшаяся часть строки:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }
  cout << "\n\n";

  // Находим последнюю запятую.
  i = s1.rfind(",");
  if(i!=string::npos) {
    cout << "Найдено соответствие в позиции " << i << endl;
    cout << "Оставшаяся часть строки:\n";
    s2.assign(s1, i, s1.size());
    cout << s2;
  }

  return 0;
}
```

Результат работы программы показан ниже.

```
Найдено соответствие в позиции: 0
Оставшаяся часть строки:
Чистые руки, горячее сердце, холодная голова

Найдено соответствие в позиции: 13
Оставшаяся часть строки:
горячее сердце, холодная голова

Найдено соответствие в позиции: 29
Оставшаяся часть строки:
холодная голова

Найдено соответствие в позиции: 27
Оставшаяся часть строки:
, холодная голова
```

Сравнение строк

Для сравнения полных строк обычно применяются перегруженные операторы сравнения, описанные выше. Однако, если возникает необходимость сравнить часть одной строки с другой строкой, необходимо использовать функцию-член **compare()**.

```
int compare(size_type start, size_type num, const string &strob) const;
```

Эта функция сравнивает *num* символов, принадлежащих объекту *strob*, с вызывающей строкой. Если вызывающая строка короче строки *strob*, функция **compare()** возвращает отрицательное число. Если вызывающая строка длиннее строки *strob*, функция **compare()** возвращает положительное число. Если длины строк равны, возвращается нуль.

Создание С-строки

Хотя объекты класса **string** полезны сами по себе, иногда возникает необходимость преобразовать их в С-строки. Например, объект класса **string** можно применить для создания имени файла. Однако при открытии файла необходимо задать указатель на обычную С-строку. Для решения этой проблемы предназначена функция **c_str()**. Ее прототип приведен ниже.

```
const char *c_str() const;
```

Эта функция возвращает указатель на С-строку, содержащуюся в вызывающем объекте класса **string**. С-строка не должна изменяться. Кроме того, нет гарантии, что она останется корректной после выполнения операций над вызывающим объектом.

Строки как контейнеры

Класс **string** соответствует многим условиям, которые предъявляются контейнерам. Это значит, что он поддерживает стандартные контейнерные функции, такие как **begin()**, **end()** и **size()**. Следовательно, к объектам класса **string** можно применять алгоритмы из библиотеки STL.

```
// Строки как контейнеры.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str1("Обработка строк в языке C++ очень проста");
  string::iterator p;
  int i;

  // Применение функции size().
  for(i=0; i<str1.size(); i++)
    cout << str1[i];
  cout << endl;

  // Применение итератора.
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;
```

```
  // Применение алгоритма count().
  i = count(str1.begin(), str1.end(), 'i');
  cout << "В строке str1 содержится" << i << "символов 'a'\n";

  // Применение алгоритма transform()
  // для преобразования строчных букв в прописные.
  transform(str1.begin(), str1.end(), str1.begin(),
            toupper);
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  return 0;
}
```

Результат работы этой программы показан ниже.

```
Обработка строк в языке C++ очень проста
Обработка строк в языке C++ очень проста
В строке str1 содержится 3 символа 'a'
ОБРАБОТКА СТРОК В ЯЗЫКЕ C++ ОЧЕНЬ ПРОСТА
```

Запись строки в другой контейнер

Хотя класс **string** является контейнерным, обычно строки хранятся в других стандартных контейнерах, например, в ассоциативных массивах или списках. Так, программу, имитирующую телефонную книжку, можно переписать иначе. Для хранения имен и телефонных номеров она использует ассоциативный массив, содержащий объекты класса **string**, а не С-строки.

```
// Применение ассоциативного массива строк
// для имитации телефонной книги.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, string> directory;

  directory.insert(pair<string, string>("Том", "555-4533"));
  directory.insert(pair<string, string>("Крис", "555-9678"));
  directory.insert(pair<string, string>("Джон", "555-8195"));
  directory.insert(pair<string, string>("Рэчел", "555-0809"));

  string s;
  cout << "Введите имя: ";
  cin >> s;

  map<string, string>::iterator p;

  p = directory.find(s);
  if(p != directory.end())
    cout << "Номер телефона: " << p->second;
  else
```

```
    cout << "Имени в справочнике нет.\n";

  return 0;
}
```

Заключительные замечания о библиотеке STL

Библиотека STL является необъемлемой частью языка C++. Многие задачи программирования можно и нужно решать с ее помощью. Эта библиотека обладает большой мощью, гибкостью, и, хотя синтаксис шаблонов довольно сложен, ее очень легко использовать. Ни один программист на языке C++ не имеет права пренебрегать библиотекой STL, поскольку она играет весьма важную роль в создании новых программ.

Полный справочник по C++

Часть III

Библиотека стандартных функций

Язык C++ содержит два вида библиотек. В первой библиотеке хранятся стандартные универсальные функции, не принадлежащие ни одному классу. Эта библиотека унаследована у языка C. Вторая библиотека содержит классы и является объектно-ориентированной. В этой части мы рассматриваем библиотеку стандартных функций, а в четвертой - библиотеку классов.

Библиотека стандартных функций разделяется на следующие части.

- Функции ввода-вывода
- Функции обработки строк и символов
- Математические функции
- Функции ввода-вывода времени и даты, а также локализации
- Функции динамического распределения памяти
- Вспомогательные функции
- Функции для работы с расширенными символами

Последняя категория функций добавлена в стандарт языка C в 1995 году и впоследствии унаследована языком C++. Она позволяет создавать расширенные (**wchar_t**) эквиваленты нескольких библиотечных функций. Честно говоря, применение расширенных символов крайне ограничено, и в языке C++ есть более эффективный способ, который описан в главе 31.

Стандарт языка C99 добавил в библиотеку функций несколько новых элементов. Некоторые из них, например, функции для работы с комплексными числами и макросы обобщенного типа, являются математическими и дублируют свойства, которые уже есть в языке C++. В любом случае библиотечные функции, определенные стандартом C99, несовместимы с языком C++. По этой причине мы не рассматриваем в нашей книге стандартную библиотеку языка C, соответствующую стандарту C99.

И последнее: все компиляторы поддерживают больше функций, чем предусмотрено стандартом языка C++. Обычно дополнительные функции предназначены для обеспечения взаимодействия между программами и операционной системой, а также для реализации других машинозависимых операций. Описание этих функций следует искать в технической документации компилятора.

Полный справочник по

Глава 25

Функции ввода-вывода языка C

В этой главе описываются функции ввода-вывода языка C. Эти функции также соответствуют стандарту языка C++. Таким образом, в программах на языке C++ нет никаких ограничений на применение функций ввода-вывода, предусмотренных в языке C. Функции, описанные в главе, впервые были определены в стандарте ANSI C, поэтому их совокупность часто называют *системой ввода-вывода ANSI C*.

Заголовочный файл, связанный с системой ввода-вывода ANSI C, называется **<cstdio>**. (Программы на языке C должны включать заголовочный файл **stdio.h**.) В этом заголовочном файле определены несколько макросов и типов, используемых файловой системой. Наиболее важным является тип **FILE**, использующийся при объявлении указателя на файл. Двумя другими типами являются типы **size_t** и **fpos_t**. Тип **size_t** (который обычно является разновидностью целого типа) определяет объект, который может содержать наибольший файл, допускаемый операционной системой. Тип **fpos_t** определяет объект, предназначенный для хранения всей информации, необходимой для однозначной идентификации любой позиции внутри файла.

Многие функции ввода-вывода задают значение встроенной глобальной целочисленной переменной **errno**. При возникновении ошибки эта переменная позволяет получить доступ к более подробной информации о возникшей ситуации. Значения переменной **errno** являются машинозависимыми.

Обзор системы ввода-вывода в языке C содержится в главах 8 и 9.

На заметку

*В этой главе описывается символьная система ввода-вывода. К ней относятся широко распространенные функции, изначально определенные в языках C и C++. В 1995 году в эту систему были включены функции, предназначенные для работы с расширенными символами (**wchar_t**). Они кратко описаны в главе 31.*

Функция clearerr

```
#include <cstdio>
void clearerr(FILE *stream);
```

Функция **clearerr()** сбрасывает флаг ошибки, связанный с потоком, на который ссылается указатель *stream*, а также обнуляет индикатор конца файла.

В случае успешного вызова функции **clearerr()** исходное значение флага ошибок, связанного с потоком, обнуляется.

Ошибки, связанные с файлами, могут возникать по разным причинам, многие из которых зависят от операционной системы. Точную природу ошибки можно определить с помощью функции **perror()**, которая выводит на экран сообщение об ошибке (см. описание функции **perror()**).

Родственные функции: **feof()**, **ferror()** и **perror()**.

Функция fclose

```
#include <cstdio>
int fclose(FILE *stream)
```

Функция **fclose()** закрывает файл, связанный с указателем *stream*, и очищает буфер. После выполнения функции **fclose()** указатель *stream* больше не связан с файлом, и любой автоматически размещенный в памяти буфер удаляется из нее.

Если функция **fclose()** выполнена успешно, она возвращает нуль, в противном случае возвращается константа **EOF**. Попытка закрыть уже закрытый файл является ошибкой. Удаление данных из памяти до закрытия файла, как и недостаток места на жестком диске, также являются ошибками.

Зависимые функции: **fopen()**, **freopen()** и **fflush()**.

Функция feof

```
#include <cstdio>
int feof(FILE *stream)
```

Функция **feof()** проверяет позицию файлового курсора, чтобы распознать конец файла, связанного с указателем *stream*. Если курсор находится в конце файла, возвращается ненулевое значение, в противном случае возвращается нуль.

После достижения конца файла последующие операции чтения вернут константу **EOF**, пока не будет вызвана функция **rewind()** или **fseek()**.

Функция **feof()** особенно полезна при работе с бинарными файлами, поскольку признак конца файла является двоичным целым числом. Для распознавания конца файла следует вызывать функцию **feof()**, а не анализировать значение, возвращаемое функцией **getc()**.

Зависимые функции: **clearerr()**, **ferror()**, **perror()**, **putc()** и **getc()**.

Функция ferror

```
#include <cstdio>
int ferror(FILE *stream)
```

Функция **ferror()** анализирует ошибку, связанную с файлом, на который ссылается указатель *stream*. Если никаких ошибок не обнаружено, возвращается нуль, в противном случае возвращается ненулевое значение.

Чтобы определить точную природу ошибки, следует вызвать функцию **perror()**.

Зависимые функции: **clearerr()**, **feof()** и **perror()**.

Функция fflush

```
#include <cstdio>
int fflush(FILE *stream);
```

Если указатель *stream* связан с файлом, открытым для записи, вызов функции **fflush()** принудительно записывает содержание буфера в файл, который остается открытым.

Если функция выполнена успешно, возвращается нуль, в противном случае возвращается константа **EOF**.

При нормальном завершении программы и переполнении все буфера очищаются автоматически. Закрытие файла также сопровождается опустошением буфера, связанного с ним, и записью его содержимого в закрывающийся файл.

Зависимые функции: **fclose()**, **fopen()**, **fread()**, **fwrite()**, **getc()** и **putc()**.

Функция fgetc

```
#include <cstdio>
int fgetc(FILE *stream);
```

Функция **fgetc()** извлекает из входного потока *stream* следующий символ и увеличивает файловый курсор на единицу. Символ считывается как значение типа **unsigned char** и преобразуется в целое число.

При обнаружении конца файла функция **fgetc()** возвращает константу **EOF**. Поскольку значение **EOF** является целым числом, при работе с бинарными файлами следует использовать функцию **feof()**. Если функция **fgetc()** обнаруживает ошибку,

она также возвращает константу **EOF**. При работе с бинарными файлами для анализа ошибок следует применять функцию **ferror()**.

Зависимые функции: **fputc()**, **getc()**, **putc()** и **fopen()**.

Функция fgetpos

```
#include <cstdio>
int fgetpos(FILE *stream, fpos_t *position);
```

Функция **fgetpos()** сохраняет в объекте, на который ссылается указатель *position*, текущее значение файлового курсора. Объект, на который ссылается указатель *position*, должен иметь тип **fpos_t**. Сохраняемое значение оказывается полезным, лишь если в дальнейшем вызывается функция **fsetpos()**.

При возникновении ошибки функция **fgetpos()** возвращает положительное число, в противном случае — нуль.

Зависимые функции: **fsetpos()**, **fseek()** и **ftell()**.

Функция fgets

```
#include <cstdio>
char *fgets(char *str, int num, FILE *stream);
```

Функция **fgets()** считывает не более *num*—1 символов из потока *stream* и помещает их в символьный массив, на который ссылается указатель *str*. Символы считываются до тех пор, пока не встретится символ перехода на новую строку либо признак конца файла, либо пока не будет достигнут заданный предел. Вслед за последним считанным символом в массив записывается нулевой байт. Символ перехода на новую строку сохраняется и становится частью массива, адресуемого указателем *str*.

В случае успеха функция **fgets()** возвращает указатель *str*, а при неудаче — нулевой указатель. Если при чтении символов происходит ошибка, содержимое массива, адресуемого указателем *str*, становится неопределенным. Поскольку нулевой указатель возвращается и при возникновении ошибки, и при достижении конца файла, для выяснения, что же произошло на самом деле, необходимо использовать функции **feof()** или **ferror()**.

Зависимые функции: **fputs()**, **fgets()**, **gets()** и **puts()**.

Функция fopen

```
#include <cstdio>
FILE *fopen(const char *fname, const char *mode);
```

Функция **fopen()** открывает файл, имя которого задается параметром *fname*, и возвращает указатель на поток, связанный с этим файлом. Вид операций, которые можно выполнять с этим файлом, задается параметром *mode*. Возможные значения параметра *mode* приведены в табл. 25.1. Имя файла должно представлять собой строку символов и не противоречить правилам, установленным операционной системой. Эта строка может также содержать спецификацию пути к файлу.

Если функция **fopen()** успешно открыла указанный файл, она возвращает указатель типа **FILE**. При неудаче возвращается нулевой указатель.

Как следует из таблицы, файл может открываться как в текстовом, так и в бинарном режиме. В текстовом режиме может происходить преобразование некоторых символов. Например, символ перехода на новую строку может преобразовываться в комбинацию кодов возврата каретки и перевода строки. В двоичном режиме подобные преобразования не выполняются.

Таблица 25.1. Возможные значения параметра mode функции fopen()

Режим	*Предназначение*
`"r"`	Открывает текстовый файл для чтения
`"w"`	Создает текстовый файл для записи
`"a"`	Записывает информацию в конец существующего текстового файла
`"rb"`	Открывает бинарный файл для чтения
`"wb"`	Создает бинарный файл для записи
`"ab"`	Записывает информацию в конец существующего бинарного файла.
`"r+"`	Открывает текстовый файл для чтения и записи
`"w+"`	Создает текстовый файл для чтения и записи
`"a+"`	Открывает текстовый файл для чтения и записи
`"rb+"` или `"r+b"`	Открывает бинарный файл для чтения и записи
`"wb+"` или `"w+b"`	Создает бинарный файл для чтения и записи
`"ab+"` или `"a+b"`	Открывает бинарный файл для чтения и записи

Следующий фрагмент программы показывает, как правильно открывать файл.

```
FILE *fp;

if ((fp = fopen("test", "w")) == NULL) {
   printf("Невозможно открыть файл.\n");
   exit();
}
```

Этот способ позволяет обнаруживать все ошибки, которые могут возникнуть во время открытия файла, например, попытку открыть файл, защищенный от записи.

Если функция **`fopen()`** открывает файл для записи, любой существующий файл с указанным именем будет стерт, а вместо него будет открыт новый файл. Если файла с таким именем не существует, он будет создан. Если файл открывается для чтения, необходимо, чтобы он уже существовал, в противном случае произойдет ошибка. Если требуется дописать информацию в конец существующего файла, необходимо использовать режим **`"a"`**. Если такого файла нет, он будет создан.

Осуществляя доступ к файлу, открытому для чтения и записи, нельзя сразу после операции записи выполнять операцию чтения (и наоборот), не вызвав одну из функций **`fflush()`**, **`fseek()`**, **`fsetpos()`** или **`rewind()`**. Исключение составляет файл, открытый для чтения, поскольку при достижении его конца операцию записи можно выполнять сразу после операции чтения.

Зависимые функции: **`fclose()`**, **`fread()`**, **`fwrite()`**, **`putc()`** и **`getc()`**.

Функция fprintf

```
#include <cstdio>
int fprintf(FILE *stream, const char *format, ...);
```

Функция **`fprintf()`** выводит в поток, связанный с указателем *stream*, значения своих аргументов в соответствии со строкой *format*. Функция возвращает количество реально выведенных символов. При возникновении ошибки возвращается отрицательное число.

Количество аргументов может изменяться от нуля до максимального значения, определенного операционной системой.

Операции преобразования, заданные в строке формата, и команды вывода идентичны операциям и командам, выполняемым функцией **`printf()`** Их полное описание содержится в разделе, посвященном функции **`printf()`**.

Зависимые функции: **`printf()`** и **`fscanf()`**.

Функция fputc

```
#include <cstdio>
int fputc(int ch, FILE *stream);
```

Функция **fputc()** записывает символ *ch* в текущую позицию потока *stream*, а затем увеличивает файловый курсор на единицу. Хотя по историческим причинам символ *ch* объявляется как переменная целого типа, функция **fputc()** преобразовывает ее в тип **unsigned char**. Поскольку все символьные аргументы преобразуются в целочисленный тип, в качестве аргументов функции **fputc()** можно использовать символы. Если аргумент имеет целый тип, его старший байт игнорируется.

В случае успешного выполнения функция **fputc()** возвращает количество считанных символов, в противном случае она возвращает константу **EOF**. Если файл открыт в бинарном режиме, константа **EOF** может представлять собой корректный символ, поэтому в этом случае для распознавании ошибки следует вызывать функцию **ferror()**.

Зависимые функции: **fgetc()**,**fopen()**, **fprintf()**, **fread()** и **fwrite()**.

Функция fputs

```
#include <cstdio>
int fputs(const char *str, FILE *stream);
```

Функция **fputs()** записывает в указанный поток содержимое строки, на которую ссылается указатель *str*. Нулевой символ, служащий индикатором конца файла, в поток не записывается.

В случае успеха функция **fputs()** возвращает положительное число, а в противном случае — константу **EOF**.

Если поток открыт в текстовом режиме, может происходить преобразование некоторых символов. Это значит, что однозначного соответствия между строкой и файлом не существует. В двоичном режиме подобные преобразования не выполняются, и строка точно соответствует содержимому файла.

Зависимые функции: **fgets()**, **gets()**, **puts()**, **fprintf()** и **fscanf()**.

Функция fread

```
#include <cstdio>
size_t fread(void *buf, size_t size, size_t count, FILE *stream);
```

Функция **fread()** считывает из потока *stream count* объектов, каждый из которых имеет размер *size* байтов, и записывает их в массив, адресуемый указателем *buf*. Файловый курсор увеличивается на число, равное количеству считанных символов.

Функция **fread()** возвращает количество действительно считанных символов. Если из потока считаны не все ожидаемые символы, значит, либо произошла ошибка, либо обнаружен конец файла. Для анализа ситуации следует вызвать функцию **feof()** или **ferror()**.

Если поток открыт в текстовом режиме, может происходить преобразование некоторых символов. Например, символ перехода на новую строку может преобразовываться в комбинацию кодов возврата каретки и перевода строки.

Зависимые функции: **fwrite()**, **fopen()**, **fscanf()**, **fgetc()** и **getc()**.

Функция freopen

```
#include <cstdio>
FILE *freopen(const char *fname, const char *mode, FILE *stream);
```

Функция **freopen()** связывает существующий поток с другим файлом. Имя нового файла задается параметром *fname*, режим доступа — параметром *mode*, а перенаправляемый поток — указателем *stream*. Параметр *mode* принимает те же значения, что и в функции **fopen()**.

При вызове функция **freopen()** сначала пытается закрыть файл, связанный с указателем *stream*. Если эта попытка оказалась неудачной, функция *freopen()* пытается открыть другой файл.

В случае успешного выполнения функция **freopen()** возвращает указатель *stream* на открытый поток, в противном случае она возвращает нулевой указатель.

Функция **freopen()** в основном используется для перенаправления системных потоков **stdin**, **stdout** и **stderr** на другие файлы.

Зависимые функции: **fopen()** и **fclose()**.

Функция fscanf

```
#include <cstdio>
int fscanf(FILE *stream, const char *format, ...);
```

Функция **fscanf()** аналогична функции **scanf()**, за исключением того, что она считывает информацию из потока, заданного указателем **stream**, а не из стандартного потока **stdin**. Подробности изложены в разделе, посвященном функции **scanf**.

Функция **fscanf()** возвращает количество аргументов, которым действительно присвоены значения. Пропущенные поля в это число не входят. Если функция возвращает константу **EOF**, значит, еще до первого присваивания произошла ошибка.

Зависимые функции: **scanf()** и **fprintf()**.

Функция fseek

```
#include <cstdio>
int fseek(FILE *stream, long offset, int origin);
```

Функция **fseek()** устанавливает позицию файлового курсора, связанного с потоком *stream*, в соответствии со значениями параметров *offset* и *origin*. Она предназначена для обеспечения произвольного доступа к файлу при выполнении операций ввода-вывода. Параметр *offset* задает количество байтов, на которое следует сместить курсор по отношению к позиции, заданной параметром *origin*. Значение параметра *origin* должно определяться одним из следующих макросов, определенных в заголовке **<cstdio>**.

Имя	*Предназначение*
SEEK_SET	Смещение отсчитывается от начала файла.
SEEK_CUR	Смещение отсчитывается от текущей позиции.
SEEK_END	Смещение отсчитывается от конца файла.

В случае успешного выполнения функция возвращает нуль, в противном случае — ненулевое значение.

Функцию **fseek()** можно использовать для перемещения файлового курсора по всему файлу. Кроме того, файловый курсор можно устанавливать на позиции, находящиеся после конца файла. Однако попытка установить файловый курсор на позицию, находящуюся перед началом файла, приводит к ошибке.

Функция **fseek()** стирает признак конца файла, связанного с указанным потоком. Более того, она аннулирует любой символ, ранее возвращенный в этот поток функцией **ungetc()**.

Зависимые функции: **ftell()**, **rewind()**, **fopen()**, **fgetpos()** и **fsetpos()**.

Функция fsetpos

```
#include <cstdio>
int fsetpos(FILE *stream, const fpos_t *position);
```

Функция **fsetpos()** перемещает файловый курсор в точку, заданную объектом, на который ссылается указатель *position*. Это значение должно быть предварительно получено с помощью вызова функции **fgetpos()**. После выполнения функции **fgetpos()** признак конца файла сбрасывается. Кроме того, она аннулирует любой символ, ранее возвращенный в этот поток функцией **ungetc()**.

Если вызов функции **fsetpos()** завершен неудачно, возвращается ненулевое число, в случае успеха функция возвращает нуль.

Зависимые функции: **fgetpos()**, **fseek()** и **ftell()**.

Функция ftell

```
#include <cstdio>
long ftell(FILE *stream);
```

Функция **ftell()** возвращает текущее положение файлового курсора, связанного с потоком *stream*. Для двоичных потоков это значение равно количеству байтов между курсором и началом файла. Для текстовых потоков возвращаемое значение может не иметь определенного смысла, если оно не является аргументом функции **fseek()**, поскольку в текстовом режиме может происходить преобразование некоторых символов, например, символ перехода на новую строку может преобразовываться в комбинацию кодов возврата каретки и перевода строки. Эти преобразования искажают реальный размер файла.

При возникновении ошибки функция **ftell()** возвращает число –1.

Зависимые функции: **fseek()** и **fgetpos()**.

Функция fwrite

```
#include <cstdio>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *stream);
```

Функция **fwrite()** записывает в поток *stream* массив, состоящий из *count* объектов, каждый из которых имеет размер *size* байтов. Массив адресуется указателем *buf*. Файловый курсор увеличивается на количество записанных символов.

Функция **fwrite()** возвращает количество реально записанных объектов. Если функция выполнена успешно, это количество равно ожидаемому. Если количество записанных объектов меньше ожидаемого, возникает ошибка.

Зависимые функции: **fread()**, **fscanf()**, **getc()** и **fgetc()**.

Функция getc

```
#include <cstdio>
int getc(FILE *stream);
```

Функция **getc()** возвращает следующий символ, считанный из входного потока, и увеличивает файловый курсор на единицу. Символ считывается как значение типа **unsigned char** и преобразуется в целое число.

При обнаружении конца файла функция **getc()** возвращает константу **EOF**. Поскольку значение **EOF** является целым числом, при работе с бинарными файлами сле-

дует использовать функцию **feof()**. Если функция **getc()** обнаруживает ошибку, она также возвращает константу **EOF**. При работе с бинарными файлами для анализа ошибок следует применять функцию **ferror()**.

Функции **getc()** и **fgetc()** идентичны. В большинстве реализаций функция **getc()** просто определяется с помощью следующего макроса.

```
#define getc(fp) fgetc(fp)
```

В этом случае макрос **getc()** заменяется вызовом функции **fgetc()**.

Зависимые функции: **fputc()**, **fgetc()**, **putc()** и **fopen()**.

Функция getchar

```
#include <cstdio>
int getchar(void);
```

Функция **getchar()** возвращает следующий символ, считанный из входного потока **stdin**. Символ считывается как значение типа **unsigned char** и преобразуется в целое число.

При обнаружении конца файла функция **getchar()** возвращает константу **EOF**. Если функция **getchar()** обнаруживает ошибку, она также возвращает константу **EOF**.

В большинстве реализаций функция **getchar()** определяется как макрос.

Зависимые функции: **fputc()**,**fgetc()**, **putc()** и **fopen()**.

Функция gets

```
#include <cstdio>
char *gets(char *str);
```

Функция**fgets()** считывает символы из потока **stdin** и помещает их в символьный массив, на который ссылается указатель *str*. Символы считываются до тех пор, пока не встретится символ перехода на новую строку либо признак конца файла. Символ перехода на новую строку не сохраняется и преобразуется в нулевой байт, служащий признаком конца строки.

В случае успеха функция **gets()** возвращает указатель *str*, а при неудаче — нулевой указатель. Если при чтении символов происходит ошибка, содержимое массива, адресуемого указателем *str*, становится неопределенным. Поскольку нулевой указатель возвращается и при возникновении ошибки, и при достижении конца файла, чтобы выяснить, что же произошло на самом деле, необходимо использовать функции **feof()** или **ferror()**.

Функция **gets()** не ограничивает количество считываемых символов, поэтому может возникнуть переполнение массива, адресуемого указателем *str*. Таким образом, функция **gets()** потенциально опасна.

Зависимые функции: **fputs()**, **fgetc()**, **fgets()** и **puts()**.

Функция perror

```
#include <cstdio>
void perror(const char *str);
```

Функция **perror()** превращает значение глобальной переменной **errno** в строку и записывает ее в поток **stderr**. Если значение указателя *str* не является нулевым, он записывается первым, а за ним после точки выводится сообщение об ошибке.

Функция printf

```
#include <cstdio>
int printf(const char format, ...);
```

Функция **printf()** записывает в поток **stdout** свои аргументы в соответствии с форматной строкой *format*.

Строка *format* может содержать элементы двух видов. К первому относятся символы, подлежащие выводу на экран. Ко второму виду — спецификаторы формата, определяющие способ представления аргументов на экране. Спецификаторы формата начинаются со знака процента, за которым следует код формата. Количество аргументов должно точно совпадать с количеством спецификаторов формата, причем их порядок следования должен быть одинаковым. Например, следующий вызов функции **printf()** выведет на экран строку "Привет f 10 всем!".

```
printf("Привет %c %d %s",'f', 10, "всем!");
```

Если количество аргументов функции **printf()** меньше количества спецификаторов формата, вывод будет неопределенным. Если число аргументов больше числа спецификаторов формата, лишние аргументы игнорируются.

Спецификаторы формата перечислены в табл. 25.2.

Таблица 25.2. Спецификаторы формата функции printf()

Код	*Формат*
`%c`	Символ
`%d`	Десятичное целое число со знаком
`%i`	Десятичное целое число со знаком
`%e`	Научный формат (строчная буква e)
`%E`	Научный формат (прописная буква E)
`%f`	Десятичное число с плавающей точкой
`%g`	В зависимости от того, какой формат короче, применяется либо `%e`, либо `%f`
`%G`	В зависимости от того, какой формат короче, применяется либо `%E`, либо `%f`
`%o`	Восьмеричное число без знака
`%s`	Строка символов
`%u`	Десятичное целое число без знака
`%x`	Шестнадцатеричное число без знака (строчные буквы)
`%X`	Шестнадцатеричное число без знака (прописные буквы)
`%p`	Указатвль
`%n`	Указатель на целочисленную переменную. Спецификатор вызывает присвоение этой целочисленной переменной количества символов, выведенных перед ним
`%%`	Знак `%`

Функция **printf()** возвращает количество фактически напечатанных символов. Отрицательное число означает ошибку.

Коды формата могут иметь модификаторы, задающие ширину поля, точность и признак выравнивания по левому краю. Целое число, указанное между знаком процента `%` и кодом формата, действует как *спецификатор минимальной ширины поля*. Он заполняет поле вывода пробелами или нулями, обеспечивая ее минимальную ширину. Если строка или число больше минимальной ширины поля, они все равно выводятся полностью. По умолчанию в качестве заполнителя используется пробел. Если пустующее поле вывода необходимо заполнить нулями, перед спецификатором ширины поля следует поставить символ 0. Например, спецификатор **%05d** заполнит нулями

пустующие позиции поля вывода, если количество цифр в целом числе, подлежащем выводу, будет меньше пяти.

Точный смысл *модификатора точности* зависит от кода модифицируемого формата. Чтобы добавить модификатор точности, после спецификатора ширины поля необходимо поставить десятичную точку и указать точность. Для форматов **e**, **E** и **f** модификатор точности означает количество цифр после десятичной точки. Например, форматный код **%10.4f** выведет на экран число, у которого количество цифр не превышает 10, четыре из которых размещаются после десятичной точки. Если модификатор применяется к спецификаторам формата **%g** или **%G**, он задает количество значащих цифр.

Если модификатор точности применяется к целым типам, он задает минимальное количество цифр, из которых должно состоять число. Если число состоит из меньшего количества цифр, оно дополняется ведущими нулями.

Если модификатор используется для вывода строк, он задает максимальную длину поля. Например, спецификатор **%5.7s** означает, что на экран будет выведена строка, состоящая как минимум из пяти символов, длина которой не превышает семи. Если строка окажется длиннее, последние символы будут отброшены.

По умолчанию вывод выравнивается по правому краю. Иначе говоря, если ширина поля больше, чем выводимые данные, результаты прижимаются к правому краю. Вывод можно выровнять по левому краю, поставив перед символом **%** знак "минус". Например, спецификатор **%-10.2f** выравнивает число с двумя знаками после точки по левому краю поля, состоящего из 10 позиций.

Для вывода значений переменных типа **short int** и **long int** функция **printf()** использует два модификатора, которые можно применять к спецификаторам **d**, **i**, **o**, **u** и **x**. Модификатор **l** (буква "эль") позволяет выводить значения типа **long int**. Например, спецификатор **%ld** означает, что на экран будет выведено значение типа **long int**. Для вывода значения типа **short int** используется модификатор **h**. Например, спецификатор **%hu** означает, что на экран будет выведено значение типа **short unsigned int**.

Начиная с 1995 года компиляторы поддерживают работу с расширенными символами. Для этого к спецификатору **c** применяется модификатор **l**. В этом случае он задает расширенный символ. Аналогично, если модификатор **l** стоит перед спецификатором **s**, на экран выводится строка расширенных символов.

Модификатор **L** можно использовать как префикс перед спецификаторами **e**, **f** и **g**. Он означает, что на экран выводится значение типа **long double**.

Спецификатор формата **%n** отличается от всех остальных. Он заставляет функцию **printf()** записывать в соответствующую переменную количество символов, уже выведенных на экран. Спецификатору **%n** должен соответствовать целочисленный указатель. После завершения функции **printf()** этот указатель будет ссылаться на переменную, в которой содержится количество символов, выведенных до спецификатора **%n**.

```
#include <stdio.h>

int main(void)
{
  int count;

  printf("Это%n проверка\n", &count);
  printf("%d", count);

  return 0;
}
```

Эта программа выведет на экран строку "Это проверка" и число 3.

Функция **printf()** имеет еще два модификатора: * и #.

Модификатор **#**, стоящий перед спецификаторами **g**, **G**, **f**, **E** или **e**, гарантирует, что число будет содержать десятичную точку, даже если оно не имеет дробной части. Если этот модификатор стоит перед спецификаторами **x** или **X** , шестнадцатеричные числа выводятся с префиксом **0x**. Если же символ **#** указан перед спецификатором **o**, число будет дополнено ведущими нулями. К другим спецификаторам модификатор **#** применять нельзя.

Ширину поля и точность представления числа можно задавать не только константами, но и переменными. Для этого вместо точных значений в спецификаторе следует указать символ *****. При сканировании строки вывода функция **printf()** поочередно сопоставляет модификатор ***** с каждым аргументом.

Зависимые функции: **scanf()** и **fprintf()**.

Функция putc

```
#include <cstdio>
int putc(int ch, FILE *stream);
```

Функция **putc()** записывает символ, содержащийся в младшем байте числа *ch*, в текущую позицию потока *stream*. Поскольку при вызове функции **putc()** все символьные аргументы преобразуются в целочисленный тип, в качестве ее аргументов можно использовать символы.

В случае успешного выполнения функция **putc()** возвращает количество считанных символов, в противном случае она возвращает константу **EOF**. Если файл открыт в бинарном режиме, константа **EOF** может представлять собой корректный символ, поэтому в этом случае для распознавания ошибки следует вызывать функцию **ferror()**.

Зависимые функции: **fgetc()**, **fputc()**, **getchar()** и **putchar()**.

Функция putchar

```
#include <cstdio>
int putchar(int ch);
```

Функция **putchar()** записывает символ, содержащийся в младшем байте числа *ch*, в текущую позицию потока **stdin**. С функциональной точки зрения эта функция эквивалентна **putc(ch, stdin)**. Поскольку при вызове функции **putchar()** все символьные аргументы преобразуются в целочисленный тип, в качестве ее аргументов можно использовать символы.

В случае успешного выполнения функция **putchar()** возвращает записанный символ, в противном случае она возвращает константу **EOF**.

Зависимая функция: **putc()**.

Функция puts

```
#include <cstdio>
int puts(const char *str);
```

Функция **puts()** записывает в поток **stdin** содержимое строки, на которую ссылается указатель *str*. Нулевой символ, служащий индикатором конца файла, преобразуется в символ перехода на новую строку.

В случае успеха функция **puts()** возвращает положительное число, а в противном случае — константу **EOF**.

Зависимые функции: **putc()**, **gets()** и **printf()**.

Функция remove

```
#include <cstdio>
int remove(const char *fname);
```

Функция **remove()** стирает файл, заданный параметром *fname*. Если файл успешно удален, функция возвращает нуль, в противном случае возвращается ненулевое значение.

Зависимая функция: **rename()**.

Функция rename

```
#include <cstdio>
int rename(const char *oldname, const char *newname)
```

Функция **rename()** меняет имя файла, заданного указателем *oldfname*, на имя, заданное параметром *newfname*. Указатель *newfname* не должен ссылаться ни на один из существующий файлов в текущем каталоге.

В случае успеха функция **rename()** возвращает нуль, в противном случае возвращается ненулевое значение.

Зависимая функция: **remove()**.

Функция rewind

```
#include <cstdio>
void rewind(FILE *stream);
```

Функция **rewind()** перемещает файловый курсор в начало указанного потока. Кроме того, она стирает признак конца файла и флаг ошибки, связанный с потоком *stream*. Функция не возвращает никакого значения.

Зависимая функция: **fseek()**.

Функция scanf

```
#include <cstdio>
int scanf(const char *format, ..);
```

Функция **scanf()** представляет собой универсальную процедуру, которая считывает данные из потока **stdin** и записывает их в переменные, указанные в списке аргументов. Она может считывать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат.

Управляющая строка состоит из символов, разделенных на три категории.

- Спецификаторы формата.
- Разделители.
- Символы, не являющиеся разделителями.

Перед спецификаторами формата стоит символ **%**. Они сообщают функции **scanf()** тип данных, подлежащих вводу. Эти коды перечислены в табл. 8.3. Например, спецификатор **%s** считывает строку, а **%d** — целое число. Спецификаторы формата сопоставляются с аргументами слева направо.

Для ввода значений типа **long int** перед спецификатором формата следует поставить модификатор **l** (букву "эль"). Для ввода значений типа **short int** перед спецификатором формата следует поставить модификатор **h**. Эти модификаторы можно ставить перед спецификаторами **d**, **i**, **o**, **u**, **x** и **n**.

Таблица 25.3. Спецификаторы формата функции scanf()

Код	Формат
%c	Символ
%d	Десятичное целое число со знаком
%i	Десятичное целое число со знаком, восьмеричное или шестнадцатеричное число
%e	Десятичное число с плавающей точкой
%f	Десятичное число с плавающей точкой
%g	Десятичное число с плавающей точкой
%o	Восьмеричное число
%s	Строка символов
%u	Десятичное целое число без знака
%x	Шестнадцатеричное число без знака (строчные буквы)
%p	Указатель
%n	Указатель на целочисленную переменную. Спецификатор вызывает присваивание этой целочисленной переменной количества символов, введенных перед ней
%[]	Набор сканируемых символов
%%	Знак %

По умолчанию спецификаторы **f**, **e** и **g** сообщают функции **scanf()**, что вводится число с плавающей точкой. Если перед этими спецификаторами стоит буква **l** ("эль"), введенное значение будет присвоено переменной типа **double**. Префикс **L** означает ввод значения типа **long double**.

Начиная с 1995 года компиляторы поддерживают работу с расширенными символами. Для этого к спецификатору **c** применяется модификатор **l**. В этом случае он задает расширенный символ, на который ссылается указатель типа **wchar_t**. Аналогично, если модификатор **l** стоит перед спецификатором **s**, на экран выводится строка расширенных символов. Кроме того, символ **l** используется для модификации набора сканируемых расширенных символов.

Разделитель в форматной строке вынуждает функцию **scanf()** пропустить один или несколько разделителей из потока ввода. В качестве разделителей используются пробел, символ табуляции и символ перехода на новую строку. По существу, один разделитель в управляющей строке вынуждает функцию **scanf()** считать, не записывая, произвольное количество разделителей, пока не будет обнаружен символ, не являющийся разделителем.

Символы, не являющиеся разделителями, вынуждают функцию **scanf()** считывать и пропускать заданные символы. Например, управляющая строка **"%d,%d"** заставляет функцию **scanf()** считать целое число, прочитать и отбросить запятую, а затем считать следующее целое число. Если указанный символ не найден, функция **scanf()** прекращает работу.

Все переменные, получающие свои значения с помощью функции **scanf()**, передаются ей по адресу. Это означает, что аргументы функции **scanf()** должны быть указателями.

Входные данные должны разделяться пробелами, символами табуляции или символами перехода на новую строку. Знаки пунктуации, такие как запятые, точки с запятой и тому подобное не считаются разделителями. Это значит, что оператор

```
scanf("%d"d", &r, &c);
```

допускает ввод чисел **10 20**, но не разрешает вводить строку **10,20**.

Можно заставить функцию **scanf()** считывать поле, не присваивая его ни одной переменной. Для этого перед кодом соответствующего формата следует поставить символ *. Например, если применить вызов

```
scanf("%d%*c%d", &x, &y);
```

к паре **10/10**, то функция **scanf()** считает число 10 и присвоит его переменной **x**, затем отбросит знак деления и присвоит число 20 переменной **y**.

В форматной строке можно указывать модификатор максимальной ширины поля. Это целое число, стоящее между символом **%** и кодом формата. Оно ограничивает количество символов, которое можно считать из поля. Например, если в переменную **address** необходимо считать не более 20 символов, следует написать следующий оператор.

```
scanf("%20s", address);
```

Если длина входной строки превышает 20 символов, следующий вызов функции **scanf()** начнет считывание с символа, на котором остановился предыдущий вызов. Если при вводе обнаружится разделитель, считывание символов завершится досрочно. В этом случае функция **scanf()** переходит к следующему полю.

Несмотря на то что пробелы, знаки табуляции и символы перехода на новую строку используются как разделители при чтении данных любых типов, при вводе отдельных символов они считываются наравне со всеми. Например, если поток ввода содержит строку **"x y"**, то фрагмент кода

```
scanf("%c%c%c", &a, &b, &c);
```

присвоит символ **x** переменной **a**, пробел — переменной **b** и символ **y** — переменной **c**.

Будьте осторожны: любые другие символы в управляющей строке, включая пробелы, знаки табуляции и символы перехода на новую строку, используются для подавления разделителей, находящихся в потоке ввода. Например, если в потоке ввода записаны символы **10t20**, вызов

```
scanf("%dt%d", &x, &y);
```

запишет число 10 в переменную **x**, а число 20 — в переменную **y**. Символ **t** будет отброшен, поскольку он указан в управляющей строке.

Функция **scanf()** поддерживает спецификатор универсального формата, называемый *набором сканируемых символов* (scanset). При обработке этого спецификатора функция будет вводить только те символы, которые входят в заданный набор. Символы записываются в массив, на который ссылается соответствующий аргумент. Для того чтобы определить набор сканируемых символов, достаточно перечислить их в квадратных скобках. Перед открывающей квадратной скобкой указывается символ процента. Например, следующий набор сканируемых символов означает, что функция **scanf()** должна вводить лишь символы **A**, **B** и **C**.

```
%[ABC]
```

При использовании набора сканируемых символов функция **scanf()** продолжает считывать символы, помещая их в соответствующий массив, пока не обнаружится символ, не входящий в заданный набор. По возвращении из функции **scanf()** массив будет содержать строку, состоящую из считанных элементов и завершающуюся нулевым байтом.

Если первым символом набора является знак **^**, набор сканируемых символов трактуется наоборот. Знак **^** сообщает функции **scanf()**, что она должна вводить только символы, *не* определенные в наборе.

В большинстве реализаций можно задавать диапазон символов, используя дефис. Например, следующий спецификатор ограничивает ввод символов диапазоном от **A** до **Z**.

```
%[A-Z]
```

Следует помнить, что набор сканируемых символов зависит от регистра. Если требуется ввести как прописные, так и строчные буквы, их следует указывать отдельно.

Функция **scanf()** возвращает количество успешно введенных полей. В это число не входят поля, которые были считаны, но не присвоены никаким переменным, поскольку были подавлены модификатором *. Если еще до первого присваивания введенного поля возникла ошибка, функция возвращает константу **EOF**.

Зависимые функции: **printf()** и **fscanf()**.

Функция setbuf

```
#include <cstdio>
void setbuf(FILE *stream, char *buf);
```

Функция **setbuf()** либо указывает буфер, который будет использоваться потоком *stream*, либо отключает буферизацию, если вызывается с нулевым аргументом. Длина буфера, определенного программистом, должна быть равна константе **BUFSIZ**, определенной в заголовке **<cstdio>**.

Функция **setbuf()** не возвращает никаких значений.

Зависимые функции: **fopen()**, **fclose()** и **setvbuf()**.

Функция setvbuf

```
#include <cstdio>
void setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Функция **setvbuf()** позволяет программисту задать буфер, его размер и режим указанного потока. В качестве буфера вывода-вывода используется символьный массив, адресуемый указателем *buf*. Размер буфера задается параметром *size*, а режим потока — параметром *mode*. Если функция **setvbuf()** вызывается с нулевым аргументом *buf*, она задает свой собственный буфер.

Параметр *mode* должен принимать значения **_IOFBF**, **_IONBF** и **IOLBF**, определенные в заголовке **<cstdio>**. Если параметр *mode* равен значению **_IOFBF**, применяется полноценная буферизация. Если параметр *mode* равен значению **_IOLBF**, используется буферизация строк. Это означает, что буфер будет очищаться каждый раз, когда в поток вывода записывается символ перехода на новую строку. Ввод буферизуется, пока не будет считан символ перехода на новую строку. Если параметр *mode* равен значению **_IONBF**, буферизация не применяется.

В случае успеха функция **setvbuf()** возвращает нуль, в противном случае она возвращает ненулевое значение.

Зависимая функция: **setbuf()**.

Функция sprintf

```
#include <cstdio>
int sprintf(char *buf, const chat *format,' ...);
```

Функция **sprintf()** идентична функции **printf()**, но вместо консоли она выводит данные в массив, адресуемый указателем *buf*. Детали описаны в разделе, посвященном функции **printf()**.

Функция возвращает значение, равное количеству символов, фактически записанных в массив.

Зависимые функции: **printf()** и **fprintf()**.

Функция sscanf

```
#include <cstdio>
int sscanf(const char *buf, const char *format, ...);
```

Функция **sscanf()** идентична функции **scanf()**, но она вводит данные из массива, адресуемого указателем *buf*, а не из стандартного потока **stdin**. Детали описаны в разделе, посвященном функции **scanf()**.

Функция возвращает значение, равное количеству фактически считанных символов. В это число не включаются пропущенные поля, заблокированные модификатором *. Если функция возвращает нуль, значит, ни одно значение не было считано. Константа **EOF** означает, что ошибка возникла еще до присвоения первого считанного значения.

Зависимые функции: **scanf()** и **fscanf()**.

Функция tmpfile

```
#include <cstdio>
FILE *tmpfile(void);
```

Функция **tmpfile()** открывает временный файл для обновления и возвращает указатель на поток. Функция автоматически использует уникальное имя файла, чтобы избежать конфликта с именами существующих файлов.

В случае успеха функция возвращает указатель на поток, а при неудаче — нулевой указатель.

После закрытия файла или прекращения работы программы временный файл, автоматически создаваемый функцией **tmpfile()**, удаляется.

Зависимая функция: **tmpnam()**

Функция tmpnam

```
#include <cstdio>
char *tmpnam(char *name);
```

Функция **tmpname()** генерирует уникальное имя файла и записывает его в массив, адресуемый указателем *name*. Длина этого массива не должна быть меньше константы **L_tmpnam**, определенной в заголовке **<cstdio>**. Основное предназначение функции **tmpnam()** — выбор имени временного файла, которое отличалось бы от любого имени, задействованного в текущем каталоге.

Функция может вызываться до **TMP_MAX** раз. Константа **TMP_MAX** определена в заголовке **<cstdio>**. Ее значение может быть больше или равно 25.

В случае успеха функция возвращает указатель на массив *name*, а при неудаче — нулевой указатель. Если значение параметра *name* равно нулю, имя временного файла сохраняется в статическом массиве, определенном в функции **tmpnam()**, возвращающей указатель на этот массив. При следующем вызове функции **tmpnam()** этот массив будет перезаписан.

Зависимая функция: **tmpfile()**.

Функция ungetc

```
#include <cstdio>
int ungetc(int ch, FILE *stream);
```

Функция **ungetc()** возвращает символ, определенный младшим байтом числа *ch*, в поток ввода *stream*. Этот символ будет считан при выполнении следующей операции ввода из потока *stream*. Вызов функции **fflush()**, **fseek()** или **rewind()** отменяет операцию **ungetc()** и отбрасывает возвращенный символ.

Гарантируется, что в поток можно вернуть один символ, однако в некоторых реализациях можно возвращать в поток несколько символов.

Символ **EOF** в поток вернуть невозможно.

Вызов функции **ungetc()** сбрасывает признак конца файла, связанного с заданным потоком. Значение курсора в текстовом файле считается неопределенным, пока не будут считаны все возвращенные символы. В этом случае файловый курсор устанавливается на позицию, которую он занимал перед первым вызовом функции **ungetc()**. Для бинарных потоков каждый вызов функции **ungetc()** уменьшает значение файлового курсора.

В случае успеха функция возвращает символ *ch*, в случае отказа — константу **EOF**.

Зависимая функция: **getc()**.

Функции vprintf, vfprintf и vsprintf

```
#include <cstdarg>
#include <cstdio>
int vprintf(char *format, va_list arg_ptr);
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
int vsprintf(char *buf, const char *format, va_list arg_ptr);
```

Функции **vprintf()**, **vfprintf()** и **vsprintf()** эквивалентны соответственно функциям **printf()**, **fprintf()** и **sprintf()**, за исключением того факта, что список аргументов заменяется указателем на список аргументов. Этот указатель должен иметь тип **va_list**, определенный в заголовке **<cstdarg>** (или заголовочный файл **stdarg.h**).

Зависимые функции: **va_arg()**, **va_start()** и **va_end()**.

Полный справочник по C++

Глава 26

Строковые и символьные функции

Стандартная библиотека обладает богатым набором функций для работы с символами и строками. Строковые функции оперируют массивами, завершающимися нулевым символом. Для работы с ними необходимо включить в программу заголовок **<cstring>**. Для символьных функций предусмотрен заголовок **<cctype>**. В программах на языке С этим заголовкам соответствуют заголовочные файлы **cstring.h** и **ctype.h**.

Поскольку ни в языке С, ни в языке С++ проверка выхода за пределы диапазона не предусмотрена, ответственность за предотвращение переполнения массива возлагается на программиста. Пренебрежение этой опасностью может закончиться крахом программы.

В языке С/С++ *печатаемым* называется символ, который можно отобразить на экране. Обычно такие символы расположены между пробелом (**0x20**) и тильдой (**0xFE**). *Управляющие символы* имеют значения, лежащие в диапазоне от **0** до **0x1F**. К ним также относится символ **DEL** (**0x7F**).

По историческим причинам параметрами символьных функций считаются целые числа, при этом используется только их младший байт. Символьные функции автоматически преобразовывают свои аргументы в тип **unsigned char**. Однако в качестве аргументов этих функций можно использовать и символы, поскольку при вызове они автоматически преобразуются в целое число.

В заголовке **<cstring>** определен тип **size_t** , который, по существу, является типом **unsigned**.

В этой главе описываются лишь функции, работающие с переменными типа **char**. Именно эти функции были изначально определены в стандартах языков С и С++ и до сих пор являются наиболее распространенными. Функции для работы с расширенными символами обсуждаются в главе 31.

Функция isalnum

```
#include <cctype>
int isalnum(int ch);
```

Если аргумент является буквой или цифрой, функция **isalnum()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимые функции: **isalpha()**, **iscntrl()**, **isdigit()**, **isgraph()**, **isprint()**, **ispunct()** и **isspace()**.

Функция isalpha

```
#include <cctype>
int isalpha(int ch);
```

Если аргумент является буквой, функция **isalpha()** возвращает ненулевое значение, в противном случае возвращается нуль. Является ли символ буквой, зависит от языка. В английском алфавите буквами считаются строчные и прописные символы от A до Z.

Зависимые функции: **isalnum()**, **iscntrl()**, **isdigit()**, **isgraph()**, **isprint()**, **ispunct()** и **isspace()**.

Функция iscntrl

```
#include <cctype>
int iscntrl(int ch);
```

Если аргумент *ch* лежит в диапазоне от нуля до **0x1F** или равен **0x7F** (**DEL**), функция **iscntrl()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимые функции: **isalnum()**, **isalpha()**, **isdigit()**, **isgraph()**, **isprint()**, **ispunct()** и **isspace()**.

Функция isdigit

```
#include <cctype>
int isdigit(int ch);
```

Если аргумент является цифрой, функция **isdigit()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимые функции: **isalnum()**, **isalpha()**, **iscntrl()**, **isgraph()**, **isprint()**, **ispunct()** и **isspace()**.

Функция isgraph

```
#include <cctype>
int isdigit(int ch);
```

Если аргумент является печатаемым символом, отличным от пробела, функция **isgraph()** возвращает ненулевое значение, в противном случае возвращается нуль. Как правило, печатаемые символы расположены в диапазоне от **0x21** до **0x7E**.

Зависимые функции: **isalnum()**, **isalpha()**, **iscntrl()**, **isdigit()**, **isprint()**, **ispunct()** и **isspace()**.

Функция islower

```
#include <cctype>
int islower(int ch);
```

Если аргумент является строчной буквой, функция **islower()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимая функция: **isupper()**.

Функция isprint

```
#include <cctype>
int isprint(int ch);
```

Если аргумент является печатаемым символом, включая пробел, функция **isprint()** возвращает ненулевое значение, в противном случае возвращается нуль. Как правило, печатаемые символы расположены в диапазоне от **0x20** до **0x7E**.

Зависимые функции: **isalnum()**, **isalpha()**, **iscntrl()**, **isgraph()**, **isdigit()**, **ispunct()** и **isspace()**.

Функция ispunct

```
#include <cctype>
int ispunct(int ch);
```

Если аргумент является знаком пунктуации, функция **ispunct()** возвращает ненулевое значение, в противном случае возвращается нуль. К знакам пунктуации относятся все печатаемые символы, не являющиеся буквами, цифрами и пробелами.

Зависимые функции: **isalnum()**, **isalpha()**, **iscntrl()**, **isdigit()**, **isgraph()** и **isspace()**.

Функция isspace

```
#include <cctype>
int isspace(int ch);
```

Если аргумент является пробелом, знаком горизонтальной или вертикальной табуляции, символом возврата каретки или перехода на новую строку, функция **isspace()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимые функции: **isalnum()**, **isalpha()**, **iscntrl()**, **isdigit()**, **isgraph()** и **ispunct()**.

Функция isupper

```
#include <cctype>
int isupper(int ch);
```

Если аргумент является прописной буквой, функция **isupper()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимая функция: **islower()**.

Функция isxdigit

```
#include <cctype>
int isxdigit(int ch);
```

Если аргумент является шестнадцатеричной цифрой, функция **isxdigit()** возвращает ненулевое значение, в противном случае возвращается нуль.

Зависимые функции: **isalnum()**, **isalpha()**, **isdigit()**, **isgraph()**, **ispunct()** и **isspace()**.

Функция memchr

```
#include <cstring>
void *memchr(const void *buffer, int ch, size_t count);
```

Функция **memchr()** ищет в массиве *buffer* первое вхождение символа *ch* среди первых *count* элементов. В случае успеха она возвращает указатель на первое вхождение символа *ch* в массив *buf*, в противном случае возвращается нулевой указатель.

Зависимые функции: **memcpy()** и **isspace()**.

Функция memcmp

```
#include <cstring>
int *memcmp(const void *buf1, const void *buf2, size_t count);
```

Функция **memcmp()** сравнивает первые *count* элементов массивов, на которые ссылаются указатели *buf1* и *buf2*.

Она возвращает целое число, интерпретация которого приведена ниже.

Значение	*Смысл*
Больше нуля	Массив *buf1* меньше массива *buf2*.
Нуль	Массив *buf1* равен массиву *buf2*.
Меньше нуля	Массив *buf1* больше массива *buf2*.

Зависимые функции: **memchr()**, **memcpy()** и **strcmp()**.

Функция memcpy

```
#include <cstring>
void *memcpy(void *to, const void *from, size_t count);
```

Функция **memcpy()** копирует первые *count* элементов массива *from* в массив *to*. Если массивы перекрываются, поведение функции **memcpy()** становится неопределенным. Она возвращает указатель на массив *to*.

Зависимая функция: **memmove()**.

Функция memmove

```
#include <cstring>
void *memmove(void *to, const void *from, size_t count);
```

Функция **memmove()** копирует первые *count* элементов массива *from* в массив *to*. Если массивы перекрываются, копирование выполняется корректно. Функция **memmove()** возвращает указатель на массив *to*.

Зависимая функция: **memcpy()**.

Функция memset

```
#include <cctype>
void *memset(void *buf, int ch, size_t count);
```

Функция **memset()** копирует младший байт символа *ch* в первые *count* символов массива *buf*. Она возвращает указатель на массив *buf*.

Зависимые функции: **memcmp()**, **memcpy()** и **memmove()**.

Функция strcat

```
#include <cstring>
char *strcat(char *str1, const char *str2);
```

Функция **strcat()** конкатенирует копию строки **str2** в строку **str1** и записывает в конец строки *str1* нулевой символ. Исходный нулевой символ, содержавшийся в строке *str1*, накрывается первым символом строки *str2*. Строка *str2* остается неизменной. Если массивы перекрываются, поведение функции **strcat()** становится неопределенным.

Функция **strcat()** возвращает указатель на строку *str1*.

Следует помнить, что проверка выхода за пределы допустимого диапазона при копировании строк не выполняется, поэтому программист должен сам гарантировать, что размер строки *str1* достаточен для хранения исходного содержимого строки *str1* и содержимого строки *str2*.

Зависимые функции: **strchr()**, **strcmp()** и **strcpy()**.

Функция strchr

```
#include <cstring>
char *strchr(const char *str, int ch);
```

Функция **strchr()** возвращает указатель на первое вхождение младшего байта числа *ch* в строку **str**. Если вхождение не обнаружено, возвращается нулевой указатель.

Зависимые функции: **strpbrk()**, **strspn()**, **strstr()** и **strtok()**.

Функция strcmp

```
#include <cstring>
int strcmp(const char *str1, const char *str2);
```

Функция **strcmp()** выполняет лексикографическое сравнение двух строк, возвращая целое число, интерпретация которого приведена ниже.

Значение	*Смысл*
Меньше нуля	Строка *str1* меньше строки *str2*.
Нуль	Строка *str1* равна строке *str2*.
Больше нуля	Строка *str1* больше строки *str2*.

Зависимые функции: **strchr()**, **strcpy()** и **strcmp()**.

Функция strcoll

```
#include <cstring>
int strcoll(const char *str1, const char *str2);
```

Функция **strcoll()** сравнивает две строки в соответствии с правилами, установленными функцией **setlocale()**.

Она возвращает целое число, интерпретация которого приведена ниже.

Значение	*Смысл*
Меньше нуля	Строка *str1* меньше строки *str2*.
Нуль	Строка *str1* равна строке *str2*.
Больше нуля	Строка *str1* больше строки *str2*.

Зависимые функции: **memcmp()** и **strcmp()**.

Функция strcpy

```
#include <cstring>
char *strcpy(char *str1, const char *str2);
```

Функция **strcpy()** копирует содержимое строки *str2* в строку *str1*. Указатель *str1* должен ссылаться на строку, завершаемую нулевым символом.

Если строки перекрываются, поведение функции **strcpy()** становится неопределенным.

Зависимые функции: **memcpy()**, **strchr()**, **strcmp()** и **strncmp()**.

Функция strcspn

```
#include <cstring>
size_t strcspn(const char *str1, const char *str2);
```

Функция **strcspn()** возвращает длину начальной подстроки строки *str1*, не содержащей символов из строки *str2*. Иначе говоря, функция **strcspn()** возвращает индекс первого символа в строке *str1*, совпадающего с каким-либо символом из строки *str2*.

Зависимые функции: **strrchr()**, **strpbrk()**, **strstr()** и **strtok()**.

Функция strerror

```
#include <cstring>
char *strerror(int errnum);
```

Функция **strerror()** возвращает указатель на строку, связанную со значением параметра *errnum*. Эта строка определяется операционной системой и не может изменяться.

Функция strlen

```
#include <cstring>
size_t strlen(const char *str);
```

Функция **strlen()** возвращает длину строки *str*, завершающейся нулем. Нулевой символ, служащий признаком конца строки, не учитывается.

Зависимые функции: **memcpy()**, **strchr()**, **strcmp()** и **strncmp()**.

Функция strncat

```
#include <cstring>
char *strcat(char *str1, const char *str2, size_t count);
```

Функция **strncat()** конкатенирует первые *count* символов строки **str2** в строку **str1** и записывает в конец строки *str1* нулевой символ. Исходный нулевой символ, содержавшийся в строке *str1*, накрывается первым символом строки *str2*. Строка *str2* остается неизменной. Если массивы перекрываются, поведение функции **strcat()** становится неопределенным.

Функция **strncat()** возвращает указатель на строку *str1*.

Следует помнить, что проверка выхода за пределы допустимого диапазона при копировании строк не выполняется, поэтому программист должен сам гарантировать, что размер строки *str1* достаточен для хранения исходного содержимого строки *str1* и содержимого строки *str2*.

Зависимые функции: **strcat()**, **strnchr()** **strncmp()** и **strncpy()**.

Функция strncmp

```
#include <cstring>
int strncmp(const char *str1, const char *str2, size_t count);
```

Функция **strncmp()** выполняет лексикографическое сравнение первых *count* символов двух строк, завершающихся нулевым байтом.

Она возвращает целое число, интерпретация которого приведена ниже.

Значение	*Смысл*
Меньше нуля	Строка *str1* меньше строки *str2*.
Нуль	Строка *str1* равна строке *str2*.
Больше нуля	Строка *str1* больше строки *str2*.

Если длина каждой из строк меньше числа *count*, сравнение заканчивается на первом попавшемся нулевом байте.

Зависимые функции: **strcmp()**, **strnchr()** и **strncpy()**.

Функция strncpy

```
#include <cstring>
char * strncpy(char *str1, const char *str2, size_t count);
```

Функция **strncpy()** копирует первые *count* символов строки *str2* в строку *str1*. Указатель *str1* должен ссылаться на строку, завершающуюся нулевым символом.

Если строки перекрываются, поведение функции **strcpy()** становится неопределенным.

Если длина строки *str2* меньше числа *count*, строка *str1* дополняется нулями. И, наоборот, если длина строки *str2* больше числа *count*, результирующая строка не содержит нулевого символа.

Функция **strncpy()** возвращает указатель на строку *str1*.

Зависимые функции: **memcpy()**, **strchr()**, **strncat()** и **strncmp()**.

Функция strpbrk

```
#include <cstring>
char *strpbrk(const char *str1, const char *str2);
```

Функция **strpbrk()** возвращает указатель на первое вхождение символа из строки *str1*, совпадающего с каким-либо символом из строки *str2*. Нулевой символ не учитывается. Если вхождение не обнаружено, возвращается нулевой указатель.

Зависимые функции: **strspn()**, **strrchr()**, **strstr()** и **strtok()**.

Функция strrchr

```
#include <cstring>
char *strrchr(const char *str, int ch);
```

Функция **strrchr()** возвращает указатель на последнее вхождение младшего байта числа *ch* в строку **str**. Если вхождение не обнаружено, возвращается нулевой указатель.

Зависимые функции: **strpbrk()**, **strspn()**, **strstr()** и **strtok()**.

Функция strspn

```
#include <cstring>
size_t strspn(const char *str1, const char *str2);
```

Функция **strspn()** возвращает длину начальной подстроки строки *str1*, содержащей только символы из строки *str2*. Иначе говоря, функция **strcspn()** возвращает индекс первого символа в строке *str1*, не совпадающего с каким-либо символом из строки *str2*.

Зависимые функции: **strpbrk()**, **strrchr()**, **strstr()** и **strtok()**.

Функция strstr

```
#include <cstring>
char *strstr(const char *str1, const char *str2);
```

Функция **strstr()** возвращает указатель на первое вхождение символа из строки *str1*, совпадающего с каким-либо символом из строки *str2*. Если вхождение не обнаружено, возвращается нулевой указатель.

Зависимые функции: **strchr()**, **strcspn()**, **strpbrk()**, **strspn()**, **strtok()** и **strrchr()**.

Функция strtok

```
#include <cstring>
char *strtok(char *str1, const char *str2);
```

Функция **strtok()** возвращает указатель на следующую лексему в строке *str1*. Символы, образующие строку *str2*, являются разделителями, определяющими лексему. Если лексемы не обнаружены, возвращается нулевой указатель.

Для того чтобы разбить строку на лексемы, сначала необходимо вызвать функцию **strtok()** и получить указатель на разбиваемую строку *str1*. В дальнейшем при вызове функции **strtok()** вместо строки *str1* следует задавать нулевой указатель. Таким образом, строка последовательно разбивается на лексемы.

В качестве разделителей в разных вызовах функции **strtok()** можно использовать различные символы.

Зависимые функции: **strchr()**, **strcspn()**, **strpbrk()**, **strrchr()** и **strspn()**.

Функция strxfrm

```
#include <cstring>
size_t *strxfrm(char *str1, const char *str2, size_t count);
```

Функция **strxfrm()** преобразует строку *str2* таким образом, чтобы ее можно было использовать для вызова функции **strcmp()**, и записывает ее в строку *str1*. После преобразования результат функции **strcmp()**, примененной к строке *str1*, совпадает с результатом функции **strcoll()**, примененной к строке *str2*. В массив *str1* записывается не более *count* символов.

Функция **strxfrm()** возвращает длину преобразованной строки.

Зависимая функция: **strcoll()**.

Функция tolower

```
#include <cctype>
int tolower(int ch);
```

Если символ *ch* является буквой, функция **tolower()** возвращает его строчный эквивалент. В противном случае символ не изменяется.

Зависимая функция: **toupper()**.

Функция toupper

```
#include <cctype>
int toupper(int ch);
```

Если символ *ch* является буквой, функция **tolower()** возвращает его прописной эквивалент. В противном случае символ не изменяется

Зависимая функция: **tolower()**.

Полный справочник по

Глава 27

Математические функции

Математические функции, входящие в состав стандартной библиотеки, разделяются на следующие категории.

- Тригонометрические
- Гиперболические
- Экспоненциальные и логарифмические
- Другие

Для всех математических функций необходим заголовок **<cmath>**. (В программах на языке C используется заголовочный файл **math.h**.) Кроме объявлений математических функций этот заголовок содержит определение макроса **HUGE_VAL**. Вместе с математическими функциями применяются также макросы **EDOM** и **ERANGE**, определенные в заголовке **<cerrno>** (или в файле **errno.h**). Если аргумент математической функции не принадлежит области ее определения, возвращается значение, зависящее от конкретной реализации, а встроенной глобальной целочисленной переменной **errno** присваивается значение константы **ERRNO**. Если значение функции превышает пределы допустимого диапазона чисел, возникает переполнение (overflow). В этом случае процедура возвращает константу **HUGE_VAL**, а переменной **errno** присваивается константа **ERANGE**, идентифицирующая ошибку. При потере значимости (underflow) функция возвращает нуль, а переменная **errno** принимает значение **ERANGE**.

Все углы задаются в радианах.

Изначально все математические функции оперировали числами типа **double**, однако стандарт C++ дополнил язык перегруженными версиями этих функций, разрешив им выполнять операции над числами, имеющими тип **float** и **long double**. Во всем остальном математические операции остались неизменными.

Функция acos

```
#include <cmath>
float acos(float arg);
double acos(double arg);
long double acos(long double arg);
```

Функция **acos()** возвращает арккосинус аргумента *arg*. Значение аргумента функции **acos()** должно лежать в интервале от -1 до 1, иначе произойдет ошибка.

Зависимые функции: **asin()**, **atan()**, **atan2()**, **sin()**, **cos()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция asin

```
#include <cmath>
float asin(float arg);
double asin(double arg);
long double asin(long double arg);
```

Функция **asin()** возвращает арксинус аргумента *arg*. Значение аргумента функции **asin()** должно лежать в интервале от -1 до 1, иначе произойдет ошибка.

Зависимые функции: **acos()**, **atan()**, **atan2()**, **sin()**, **cos()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция atan

```
#include <cmath>
float atan(float arg);
double atan(double arg);
long double atan(long double arg);
```

Функция **atan()** возвращает арктангенс аргумента *arg*.

Зависимые функции: **asin()**, **acos()**, **atan2()**, **sin()**, **cos()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция atan2

```
#include <cmath>
float atan2(float x, float y);
double atan2(double x, doubel y);
long double atan2(long double x, long double y);
```

Функция **atan2()** возвращает арктангенс значения *y/x*. Эта функция использует знаки своих аргументов для определения квадранта, которому принадлежит ее возвращаемое значение.

Зависимые функции: **asin()**, **acos()**, **atan()**, **sin()**, **cos()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция ceil

```
#include <cmath>
float ceil(float num);
double ceil(double num);
long double ceil(long double num);
```

Функция **ceil()** возвращает ближайшее целое число (представленное как действительное число с плавающей точкой), которое не меньше значения аргумента *num*. Например, если значение *num* равно 1.02, функция **ceil()** вернет число 2.0. (Эта операция называется округлением с избытком. — *Прим. ред.*)

Зависимые функции: **floor()** и **fmod()**.

Функция cos

```
#include <cmath>
float cos(float arg);
double cos(double arg);
long double cos(long double arg);
```

Функция **cos()** возвращает косинус аргумента *arg*. Значение аргумента должно быть выражено в радианах.

Зависимые функции: **asin()**, **acos()**, **atan()**, **atan2()**, **sin()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция cosh

```
#include <cmath>
float cosh(float arg);
double cosh(double arg);
long double cosh(long double arg);
```

Функция **cosh()** возвращает гиперболический косинус аргумента *arg*.

Зависимые функции: **asin()**, **acos()**, **atan2()**, **atan()**, **sin()**, **tan()**, **cos()**, **cosh()** и **tanh()**.

Функция exp

```
#include <cmath>
float exp(float arg);
double exp(double arg);
long double exp(long double arg);
```

Функция **exp()** возвращает основание натурального логарифма *e*, возведенное в степень *arg*.

Зависимая функция: **log()**.

Функция fabs

```
#include <cmath>
float fabs(float num);
double fabs(double num);
long double fabs(long double num);
```

Функция **fabs()** возвращает абсолютное значение аргумента *num*.

Зависимая функция: **abs()**.

Функция floor

```
#include <cmath>
float floor(float num);
double floor(double num);
long double floor(long double num);
```

Функция **floor()** возвращает набольшее целое число (представленное как действительное число с плавающей точкой), не превышающее значения аргумента *num*. Например, если значение *num* равно 1.02, функция **floor()** вернет число 1.0. (Эта операция называется округлением с недостатком. — *Прим. ред.*)

Зависимые функции: **fceil()** и **fmod()**.

Функция fmod

```
#include <cmath>
float fmod(float x, float y);
double fmod(double x, double y);
long double fmod(long double x, long double y);
```

Функция **fmod()** возвращает остаток от деления *x*/*y*.

Зависимые функции: **ceil()**, **floor()** и **fabs()**.

Функция frexp

```
#include <cmath>
float frexp(float num, int *exp);
double frexp(double num, int *exp);
long double frexp(long double num, int *exp);
```

Функция **frexp()** раскладывает число *num* на мантиссу, изменяющуюся в диапазоне от 0.1 до 1, не включая 1, и целый показатель степени, так что $num=mantissa*2^{exp}$. Функция возвращает мантиссу, а значение показателя степени сохраняется в переменной, на которую ссылается указатель *exp*.

Зависимая функция: **ldexp()**.

Функция ldexp

```
#include <cmath>
float ldexp(float num, int exp);
double ldexp(double num, int exp);
long double ldexp(long double num, int exp);
```

Функция **ldexp()** возвращает число $num*2^{exp}$. Если возникает ошибка, функция возвращает значение **HUGE_VAL**.

Зависимые функции: **frexp()** и **modf()**.

Функция log

```
#include <cmath>
float log(float num);
double log(double num);
long double log(long double num);
```

Функция **log()** возвращает натуральный логарифм числа *num*. Если аргумент *num* отрицателен или равен нулю, возникает ошибка.

Зависимая функция: **log10()**.

Функция log10

```
#include <cmath>
float log10(float num);
double log10(double num);
long double log10(long double num);
```

Функция **log10()** возвращает десятичный логарифм числа *num*. Если аргумент *num* отрицателен или равен нулю, возникает ошибка.

Зависимая функция: **log()**.

Функция modf

```
#include <cmath>
float modf(float num, float *i);
double modf(double num, float *i);
long double modf(long double num, int *i);
```

Функция **modf()** раскладывает число *num* на целую и дробную части. Функция возвращает дробную часть, а целая часть сохраняется в переменной, на которую ссылается указатель *i*.

Зависимые функции: **frexp()** и **ldexp()**.

Функция pow

```
#include <cmath>
float pow(float base, float exp);
```

```
float pow(double base, int exp);
double pow(double base, double exp);
double pow(double base, int exp);
long double pow(long double num, long double exp);
long double pow(long double num, int exp);
```

Функция **pow()** возвращает число *base*, возведенное в степень *exp*. Если основание степени *base* равно нулю, а показатель степени *exp* меньше или равен нулю, может произойти ошибка, связанная с выходом аргумента из области определения функции (domain error). Эта ошибка возникает также, если аргумент *base* отрицателен, а аргумент *exp* не является целым числом. В случае переполнения возникает ошибка, связанная с выходом за пределы допустимых значений (range error).

Зависимые функции: **exp()**, **log()** и **sqrt()**.

Функция sin

```
#include <cmath>
float sin(float arg);
double sin(double arg);
long double sin(long double arg);
```

Функция **sin()** возвращает синус аргумента *arg* Значение аргумента должно быть выражено в радианах

Зависимые функции: **asin()**, **acos()**, **atan()**, **atan2()**, **cos()**, **tan()**, **sinh()**, **cosh()** и **tanh()**.

Функция sinh

```
#include <cmath>
float sinh(float arg);
double sinh(double arg);
long double sinh(long double arg);
```

Функция **sinh()** возвращает гиперболический синус аргумента *arg*.

Зависимые функции: **asin()**, **acos()**, **atan()**, **atan2()**, **sin()**, **cos()**, **tan()**, **cosh()** и **tanh()**.

Функция sqrt

```
#include <cmath>
float sqrt(float num);
double sqrt(double num);
long double sqrt(long double num);
```

Функция **sqrt()** возвращает квадратный корень аргумента *num*. Если значение аргумента отрицательно, возникает ошибка, связанная с выходом из области определения функции.

Зависимые функции: **exp()**, **log()** и **pow()**.

Функция tan

```
#include <cmath>
float tan(float arg);
double tan(double arg);
long double tan(long double arg);
```

Функция **tan()** возвращает тангенс аргумента *arg*. Значение аргумента должно быть выражено в радианах.

Зависимые функции: **acos()**, **asin()**, **atan()**, **atan2()**, **cos()**, **sin()**, **sinh()**, **cosh()** и **tanh()**.

Функция tanh

```
#include <cmath>
float tanh(float arg);
double tanh(double arg);
long double tanh(long double arg);
```

Функция **tanh()** возвращает гиперболический тангенс аргумента *arg*.

Зависимые функции: **acos()**, **asin()**, **atan()**, **atan2()**, **cos()**, **sin()**, **tan()**, **sinh()** и **cosh()**.

Полный справочник по

Глава 28

Функции времени, даты и локализации

В стандартной библиотеке содержатся несколько функций, предназначенных для работы со временем и датами. Кроме того, в ней определены функции, обрабатывающие геополитическую информацию, связанную с программой. Глава посвящена описанию этих функций.

Для работы с функциями времени и даты необходим заголовок **<ctime>**. В этом заголовке определены три типа данных, связанных с представлением времени: **clock_t**, **time_t** и **tm**. Типы **clock_t** и **time_t** предназначены для представления системного времени и даты в виде некоторого целого значения, которое называется *календарным временем*. Структура **tm** хранит дату и время, разбитые на компоненты. Вот как выглядит ее определение.

```
struct tm {
  int tm_sec;     /* секунды, 0-61 */
  int tm_min;     /* минуты, 0-59  */
  int tm_hour;    /* часы, 0-23 */
  int tm_day;     /* день месяца, 1-31 */
  int tm_mon;     /* месяцы, начиная с января, 0-11 */
  int tm_year;    /* годы, начиная с 1900 */
  int tm_wday;    /* дни, начиная с воскресенья, 0-6 */
  int tm_yday;    /* дни, начиная с 1 января, 0-365 */
  int tm_isdst;   /* индикатор летнего времени */
}
```

Если в текущий момент действует летнее время, значение поля **tm_isdst** положительно, в противном случае оно равно нулю. Эта форма представления времени и даты называется *покомпонентной* (broken-down time).

Кроме того, в заголовке **<ctime>** определен макрос **CLOCKS_PER_SEC**, задающий количество тактов системных часов в секунду.

Функции, учитывающие геополитическое окружение, объявлены в заголовке **<clocale>**. (В программах на языке C для этого используется заголовочный файл **locale.h**.)

Функция asctime

```
#include <ctime>
char *asctime(const struct tm *ptr);
```

Функция **asctime()** возвращает указатель на строку, содержащую информацию, хранящуюся в структуре, на которую ссылается указатель *ptr*. Строка преобразуется следующим образом.

день месяц дата часы:минуты:секунды год \n \0

Например:

```
Fri Apr 15 12:05:34 2005
```

Указатель на структуру, передаваемый функции **asctime()**, обычно получают с помощью функций **localtime()** или **gmtime()**.

Буфер, используемый функцией **asctime()** для хранения форматированной строки, представляет собой статический символьный массив и перезаписывается при каждом вызове. Если содержимое этой строки необходимо сохранить, ее следует скопировать.

Зависимые функции: **localtime()**, **gmtime()**, **time()** и **ctime()**.

Функция clock

```
#include <ctime>
clock_t clock(void);
```

Функция **clock()** возвращает приблизительное время работы вызывающей программы. Чтобы выразить это значение в секундах, следует поделить его на константу **CLOCK_PER_SEC**. Если время определить невозможно, функция возвращает число -1.

Зависимые функции: **time()**, **asctime()** и **ctime()**.

Функция ctime

```
#include <ctime>
char *ctime(const time_t *time);
```

Получив указатель на структуру, содержащую календарное время, функция **ctime()** возвращает указатель на строку, имеющую следующий вид.

день месяц дата часы:минуты:секунды год \n \0

Кроме того, структуру, содержащую календарное время, можно получить с помощью функции **time()**.

Буфер, используемый функцией **ctime()** для хранения форматированной строки, представляет собой статический символьный массив и перезаписывается при каждом вызове. Для того чтобы сохранить содержимое этой строки, ее следует скопировать.

Зависимые функции: **localtime()**, **gmtime()**, **time()** и **asctime()**.

Функция difftime

```
#include <ctime>
double difftime(time_t time2, time_t time1);
```

Функция **difftime()** возвращает разность между аргументами *time1* и *time2*, выраженную в секундах, иначе говоря, величину *time2-time1*.

Зависимые функции: **localtime()**, **gmtime()**, **time()** и **asctime()**.

Функция gmtime

```
#include <ctime>
struct tm *gmtime(const time_t *time);
```

Функция **gmtime()** возвращает указатель на покомпонентную структуру типа **tm**, в котором записано всеобщее скоординированное время (Universal Coordinated Time — UTC), которое часто называют временем по Гринвичу. Значение *time* обычно получают с помощью функции **time()**. Если система не поддерживает всеобщее скоординированное время, возвращается указатель **NULL**.

Структура, которую функция **gmtime()** использует для покомпонентного представления времени, представляет собой статический символьный массив и перезаписывается при каждом вызове. Для того чтобы сохранить содержимое этой строки, ее следует скопировать.

Зависимые функции: **localtime()**, **time()** и **asctime()**.

Функция localeconv

```
#include <clocale>
struct lconv *localeconv(void);
```

Функция **localeconv()** возвращает указатель на структуру типа **lconv**, содержащую разнообразную информацию о геополитическом окружении, которое влияет на способ форматирования чисел. Эта структура организована следующим образом.

```
struct lconv {
  char *decimal_point;      /* Символ десятичной точки
                               для значений, не имеющих
```

```
                                денежного смысла. */
  char *thousands_sep;      /* Разделитель тысяч для
                               значений, не имеющих
                               денежного смысла. */
  char *grouping;           /* Определяет способ группировки
                               значений, не имеющих
                               денежного смысла. */
  char *int_curr_symbol;    /* Символ международной валюты. */
  char *currency_symbol;    /* Символ национальной валюты. */
  char *mon_decimal_point   /* Символ десятичной точки
                               для значений, имеющих
                               денежный смысл. */
  char *mon_thousands_sep; /* Разделитель тысяч для
                               значений, имеющих
                               денежный смысл. */
  char *mon_grouping;       /* Определяет способ группировки
                               значений, имеющих
                               денежный смысл. */
  char *positive_sign;      /* Индикатор положительных значений,
                               имеющих денежный смысл. */
  char *negative_sign;      /* Индикатор отрицательных значений,
                               имеющих денежный смысл. */
  char int_frac_digits;     /* Количество цифр после десятичной
                               точки в представлении значений,
                               имеющих денежный смысл,
                               в международном формате. */
  char frac_digits;         /* Количество цифр после десятичной
                               точки в представлении значений,
                               имеющих денежный смысл,
                               в национальном формате. */
  char p_cs_precedes;       /* Равно 1, если символ валюты
                               предшествует положительному
                               значению, и 0, если символ
                               валюты следует за значением. */
  char p_sep_by_space;      /* Равно 1, если символ валюты
                               отделяется от значения пробелом,
                               и 0 — в противном случае. */
  char n_cs_precedes;       /* Равно 1, если символ валюты
                               предшествует отрицательному
                               значению, и 0, если символ
                               валюты следует за значением. */
  char n_sep_by_space;      /* Равно 1, если символ валюты
                               отделяется от отрицательного
                               значения пробелом,
                               и 0 — в противном случае. */
  char p_sign_posn;         /* Указывает позицию символа
                               положительного значения. */
  char n_sign_posn;         /* Указывает позицию символа
                               отрицательного значения. */
```

Функция **localeconv()** возвращает указатель на структуру типа **lconv**. Содержимое этой структуры изменять нельзя. Специфическая информация о структуре типа **lconv**, зависящая от реализации, содержится в документации компилятора.

Зависимая функция: **setlocale()**.

Функция localetime

```
#include <ctime>
struct tm *localetime(const time_t *time);
```

Функция **localetime()** возвращает указатель на покомпонентную структуру типа **tm**, в котором записано локальное время. Значение *time* обычно получают с помощью функции **time()**.

Структура, которую функция **localetime()** использует для покомпонентного представления времени, представляет собой статический символьный массив и перезаписывается при каждом вызове. Для того чтобы сохранить содержимое этой строки, ее следует скопировать.

Зависимые функции: **gmtime()**, **time()** и **asctime()**.

Функция mktime

```
#include <ctime>
time_t mktime(struct tm *time);
```

Функция **mktime()** возвращает календарный эквивалент покомпонентного представления времени, записанного в структуре, на которую ссылается указатель *time*. Элементы **tm_wday** и **tm_yday** задаются функцией, поэтому их не обязательно определять перед вызовом.

Если функция **mktime()** не может предоставить информацию о корректном календарном времени, она возвращает значение -1.

Зависимые функции: **time()**, **gmtime()**, **asctime()** и **ctime()**.

Функция setlocale

```
#include <clocale>
char *setlocale(int type, const char *locale);
```

Функция **setlocale()** задает значения некоторых параметров, чувствительных к геополитическому окружению, в котором выполняется программа. Если параметр *locale* является нулевым указателем, функция возвращает указатель на строку текущей локализации. В противном случае функция **setlocale()** пытается использовать строку, на которую ссылается указатель *locale*, для установки параметров локализации в соответствии с индикатором *type*. Строки локализации, поддерживаемые компилятором, обычно приводятся в документации.

В момент вызова индикатор *type* должен быть равен одной из следующий констант.

```
LC_ALL
LC_COLLATE
LC_TYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

Макрос **LC_ALL** относится ко всем категориям локализации. Макрос **LC_COLLATE** влияет на выполнение функции **strcoll()**, **LC_CTYPE** изменяет характер работы символьных функций, а **LC_MONETARY** определяет денежный формат. Макрос **LC_NUMERIC** изменяет символ десятичной точки для функций форматированного ввода-вывода, а **LC_TIME** определяет работу функции **strftime()**.

Функция **setlocale()** возвращает указатель на строку, связанную с параметром *type*.

Зависимые функции: **localeconv()**, **time()**, **strcoll()** и **strftime()**.

Функция strftime

```
#include <ctime>
size_t strftime(char *str, size_t maxsize,
                const char *fmt, const struct tm *time);
```

Функция **strftime()** записывает время, дату и другую информацию в строку *str* в соответствии с командами форматирования, заданными в строке *fmt*, используя структуру покомпонентного представления времени *time*. Максимальное количество символов, которое можно записать в строку *str*, задается параметром *maxsize*.

Функция **strftime()** аналогична функции **sprintf()**. Она распознает команды форматирования, начинающиеся символом **%**, и записывает форматированный результат в строку. Команды форматирования задают точный способ представления информации о времени и дате в строке *str*. Любые другие символы, содержащиеся в строке форматирования, копируются в строку *str* без изменения. Время и дата отображаются в соответствии с локальными установками. Команды форматирования приведены ниже. Обратите внимание на то, что многие команды различают прописные и строчные буквы.

Функция **strftime()** возвращает количество символов, записанных в строку *str*. При неудаче она возвращает нуль.

Команда форматирования	*Подстановка*
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Стандартная строка даты и времени
%d	День месяца в виде десятичного числа (0–31)
%H	Час (0–23)
%I	Час (1–12)
%j	День года в виде десятичного числа (1–366)
%m	Месяц в виде десятичного числа (1–12)
%M	Минута в виде десятичного числа (0–59)
%p	Локальный эквивалент обозначений AM (до полудня) и PM (после полудня)
%S	Секунда в виде десятичного числа (0–60)
%U	Неделя года, считая воскресенье первым днем (0–53)
%w	День недели в виде десятичного числа (0–6, воскресенье задается нулем)
%W	Неделя года, учитывая, что первый день — понедельник (0–53)
%x	Стандартная строка даты
%X	Стандартная строка времени
%y	Год в виде десятичного числа, столетие игнорируется (0–99)
%Y	Год в виде десятичного числа, столетие учитывается
%Z	Название часового пояса
%%	Знак процента

Функция time

```
#include <ctime>
time_t time(time_t *time);
```

Функция **time()** возвращает текущее календарное время, установленное системой. Если системное время не задано, функция возвращает число -1.

Функцию **time()** можно вызывать либо с нулевым указателем, либо с указателем на переменную типа **time_t**. Во втором случае переменной будет присвоено календарное время.

Зависимые функции: **localtime()**, **gmtime()**, **stftime()** и **ctime()**.

Полный справочник по

Глава 29

Функции динамического распределения памяти

В этой главе описываются функции динамического распределения памяти, унаследованные от языка C. Их ядро образуют функции **malloc()** и **free()**. При каждом вызове функции **malloc()** выделяется очередной участок динамической памяти. Каждый вызов функции **free()** освобождает ранее выделенную память. Область динамической памяти называется *кучей*. Прототипы функций динамического распределения памяти находятся в заголовке **<cstdlib>**. Программы на языке C должны вместо него использовать заголовочный файл **stdlib.h**.

Все компиляторы языка C++ поддерживают четыре функции динамического распределения памяти: **calloc()**, **malloc()**, **free()** и **realloc()**. Однако каждый компилятор поддерживает также их варианты, учитывающие специфику конкретной операционной среды. Детали следует искать в документации компилятора.

Несмотря на то что функции, описываемые в этом разделе, являются частью языка C++, мы не рекомендуем применять их в своих программах, поскольку в языке C++ предусмотрены операторы динамического распределения памяти **new** и **delete**. Они обладают рядом преимуществ по сравнению с функциями динамического распределения памяти. Во-первых, оператор **new** автоматически определяет правильный объем памяти, необходимый для размещения объекта указанного типа. Во-вторых, он возвращает указатель правильного типа, ссылающийся на выделенную память. В-третьих, операторы **new** и **delete** можно перегружать. Благодаря этим особенностям операторы **new** и **delete** вытеснили функции динамического распределения памяти из программ, написанных на языке C++.

Функция calloc

```
#include <cstdlib>
void *calloc(size_t num, size_t size);
```

Функция **calloc()** выделяет память, размер которой равен *num*size* байтов. Иначе говоря, функция **calloc()** выделяет динамическую память, достаточную для размещения массива, состоящего из *num* элементов, имеющих размер *size* байтов.

Функция **calloc()** возвращает указатель на первый байт выделенной области. Если требуемый объем памяти выделить невозможно, возвращается нулевой указатель. Прежде чем использовать выделенную память, необходимо проверить значение, возвращаемое функцией **calloc()**.

Зависимые функции: **free()**, **malloc()** и **realloc()**.

Функция free

```
#include <cstdlib>
void free(void *ptr);
```

Функция **free()** освобождает память, отмеченную указателем *ptr*. После этого освобожденную память можно использовать вновь.

Функцию **free()** можно применять лишь к указателю, использованному ранее в вызове функций **malloc()** или **calloc()**. В противном случае функция **free()** может разрушить весь механизм распределения динамической памяти и вызвать крах операционной системы.

Зависимые функции: **calloc()**, **malloc()** и **realloc()**.

Функция malloc

```
#include <cstdlib>
void *malloc(size_t size);
```

Функция **malloc()** возвращает указатель на первый байт выделенной области, имеющей размер *size* байтов. Если требуемый объем памяти выделить невозможно, возвращается нулевой указатель. Прежде чем использовать выделенную память, необходимо проверить значение, возвращаемое функцией **malloc()**. Попытка использовать нулевой указатель обычно приводит к краху операционной системы.

Зависимые функции: **free()**, **calloc()** и **realloc()**.

Функция realloc

```
#include <cstdlib>
void *realloc(void *ptr, size_t size);
```

Функция **realloc()** изменяет размер выделенной ранее памяти, на которую ссылается указатель *ptr*. Значение параметра *size* может быть как меньше, так и больше предыдущего объема. Функция **realloc()** возвращает указатель на первый байт вновь выделенного блока памяти, поскольку может возникнуть необходимость скопировать его в другое место. В этом случае содержимое старого блока копируется в новый, и потери информации не происходит.

Если указатель *ptr* является нулевым, функция **realloc()** просто выделяет *size* байтов памяти и возвращает указатель на первый байт выделенного участка. Если параметр *size* равен нулю, память, на которую ссылается указатель *ptr*, освобождается.

Если требуемый объем памяти выделить невозможно, возвращается нулевой указатель, и старый блок памяти остается неизменным.

Зависимые функции: **free()**, **malloc()** и **calloc()**.

Полный справочник по C++

Глава 30

Служебные функции

В стандартной библиотеке определен ряд служебных функций. К ним относятся функции, выполняющие преобразования, обработку списков аргументов, сортировку и поиск, а также генерирующие случайные числа. Многие из этих функций объявлены в заголовке **<cstdlib>**. (В программах на языке C ему соответствует заголовочный файл **stdlib.h**.) Кроме того, в этом заголовке определены типы **div_t** и **ldiv_t**, являющиеся типами значений, возвращаемых функциями **div()** и **ldiv()** соответственно. Помимо этого, в заголовке **<cstdio>** определен тип **size_t**, представляющий собой тип значения, возвращаемого оператором **sizeof**. В заголовке определены следующие макросы.

Макрос	*Значение*
MB_CUR_MAX	Максимальная длина многобайтового символа (в байтах).
NULL	Нулевой указатель.
RAND_MAX	Максимальное значение, которое может вернуть функция **rand()**.
EXIT_FAILURE	Значение, возвращаемое вызывающему процессу в случае аварийного завершения программы.
EXIT_SUCCESS	Значение, возвращаемое вызывающему процессу в случае успешного завершения программы.

Если для выполнения функции необходим другой заголовок, этот факт отмечается в соответствующем разделе.

Функция abort

```
#include <cstdlib>
void abort(void);
```

Функция **abort()** приводит к аварийному завершению программы. Как правило, при этом буфера файлов не выгружаются. Если операционная система позволяет, функция **abort()** возвращает вызывающему процессу значение, зависящее от конкретной реализации и означающее аварийное завершение работы.

Зависимые функции: **exit()** и **atexit()**.

Функция abs

```
#include <cstdlib>
int abs(int num);
long abs(long num);
double abs(double num);
```

Функция **abs()** возвращает абсолютное значение числа *num*. Версия функции **abs()** для чисел типа **long** совпадает с функцией **labs()**. Версия функции **abs()** для чисел типа **double** совпадает с функцией **fabs()**.

Зависимая функция: **labs()**.

Макрос assert

```
#include <cassert>
void assert(int exp);
```

Если значение выражения *exp* равно нулю, макрос **assert()**, определенный в заголовке **<cassert>**, записывает в поток **stderr** информацию об ошибке, а затем пре-

кращает выполнение программы. В противном случае макрос **assert()** не выполняет никаких действий. Хотя точное содержание сообщений зависит от конкретной реализации, многие компиляторы используют следующий шаблон.

```
Assertion failed: <выражение>, file <файл>, line <linenum>
```

Макрос **assert()** обычно используется для верификации программы, а *выражение* составляется так, чтобы оно принимало значение **true**, только если никаких ошибок не произошло.

Макросы **assert()** не обязательно удалять из программы после отладки, поскольку, если в программе определен макрос **NDEBUG**, макрос **assert()** просто игнорируется.

Зависимая функция: **abort()**.

Функция atexit

```
#include <cstdlib>
int atexit(void (*func)(void));
```

Функция **atexit()** регистрирует функцию, которая должна выполняться при выходе из программы. Функция **atexit()** возвращает нуль, если функция выхода успешно зарегистрирована, и ненулевое значение — в противном случае.

В качестве функции выхода можно зарегистрировать до 32 функций, и все они вызываются в порядке, обратном порядку их регистрации.

Зависимые функции: **exit()** и **abort()**.

Функция atof

```
#include <cstdlib>
double atof(const char *str);
```

Функция **atof()** преобразует строку *str* в значение типа **double**. Строка должна содержать корректное представление числа с плавающей точкой. В противном случае возвращаемое значение не определено.

Число может завершаться любым символом, недопустимым для представления действительных чисел, т.е. разделителем, знаком пунктуации (но не точкой) и символом, отличным от букв **E** и **e**. Это значит, что при вызове **atof("100.00HELLO")** функция вернет число 100.00.

Зависимые функции: **atoi()** и **atol()**.

Функция atoi

```
#include <cstdlib>
int atoi(const char *str);
```

Функция **atoi()** преобразует строку *str* в значение типа **int**. Строка должна содержать корректное представление целого числа. В противном случае возвращаемое значение не определено.

Число может завершаться любым символом, недопустимым для представления целых чисел, т.е. разделителем, знаком пунктуации и символом. Это значит, что при вызове **atoi("123.23")** функция вернет число 123, а дробная часть ".23" будет проигнорирована.

Зависимые функции: **atof()** и **atol()**.

Функция atol

```
#include <cstdlib>
long atol(const char *str);
```

Функция **atol()** преобразует строку *str* в значение типа **long**. Строка должна содержать корректное представление длинного целого числа. В противном случае возвращаемое значение не определено, однако, как правило, в этих случаях возвращается 0.

Число может завершаться любым символом, недопустимым для представления целых чисел, т.е. разделителем, знаком пунктуации и символом. Это значит, что при вызове **atoi("123.23")** функция вернет число 123, а дробная часть ".23" будет проигнорирована.

Зависимые функции: **atof()** и **atoi()**.

Функция bsearch

```
#include <cstdlib>
void *bsearch(const void *key, const void *buf,
              size_t num, size_t size,
              int (*compare)(const void *, const void *));
```

Функция **bsearch()** выполняет бинарный поиск элемента в упорядоченном массиве *buf* и возвращает указатель на первый элемент, соответствующий ключу *key*. Количество элементов в массиве задается параметром *num*, а размер каждого элемента (в байтах) задается параметром **size**.

Для сравнения элементов массива с заданным ключом используется функция, адресуемая указателем *compare*. Объявление функции **compare()** должно выглядеть следующим образом.

```
int func_name(const void *arg1, const void *arg2);
```

Она должна возвращать значения, описанные в слеdующей таблице.

Сравнение	*Возвращаемое значение*
arg1 меньше *arg2*	Меньше нуля
arg1 равно *arg2*	Нуль
arg1 больше *arg2*	Больше нуля

Массив должен быть упорядочен в порядке возрастания.

Если в массиве нет элемента, соответствующего заданному ключу, возвращается нулевой указатель.

Зависимая функция: **qsort()**.

Функция div

```
#include <cstdlib>
div_t div(int numerator, int denominator);
ldiv_t div(long numerator, long denominator);
```

Версия функции **div()** для аргументов типа **int** возвращает частное и остаток от деления числителя *numerator* на знаменатель *denominator* в виде структуры типа **div_t** . Версия функции **div()** для аргументов типа **long** возвращает частное и остаток в ви-

де структуры типа **ldiv_t**. Версия функции **div()** для аргументов типа **long** эквивалентна функции **ldiv()**.

Структура типа **div_t** содержит по крайней мере два поля

```
int quot; /* Частное /*
int rem;  /* Остаток */
```

Структура типа **ldiv_t** содержит по крайней мере два поля.

```
int quot; /* Частное /*
int rem;  /* Остаток */
```

Зависимая функция: **ldiv()**.

Функция exit

```
#include <cstdlib>
void exit(int exit_code);
```

Функция **exit()** вызывает немедленное прекращение работы программы.

Вызывающему процессу, в роли которого, как правило, выступает операционная система, возвращается значение *exit_code*. По соглашению, если значение *exit_code* равно нулю (константе **EXIT_SUCCESS**), предполагается, что программа завершилась нормально. Ненулевое значение (константа **EXIT_FAILURE**) используется для индикации ошибки.

Зависимые функции: **atexit()** и **abort()**.

Функция getenv

```
#include <cstdlib>
char *getenv(const char *name);
```

Функция **getenv()** возвращает указатель на данные, связанные со строкой *name* в таблице, содержащей информацию об окружении. Возвращаемая строка не должна изменяться в программе.

Окружением программы считаются, например, пути к различным файлам и подключенные устройства.

Если функция **getenv()** вызывается с аргументом, который не совпадает ни с одной из характеристик окружения, возвращается нулевой указатель.

Функция labs

```
#include <cstdlib>
long labs(int num);
```

Функция **labs()** возвращает абсолютное значение числа *num*.

Зависимая функция: **abs()**.

Функция ldiv

```
#include <cstdlib>
ldiv_t ldiv(long numerator, long denominator);
```

Функция **ldiv()** для аргументов типа **int** возвращает частное и остаток от деления числителя *numerator* на знаменатель *denominator* в виде структуры типа **ldiv_t** .

Структура типа **ldiv_t** содержит по крайней мере два поля.

```
int quot; /* Частное /*
int rem;  /* Остаток */
```

Зависимая функция: **div()**.

Функция longjmp

```
#include <csetjmp>
void longjmp(jmp_buf envbuf, int status);
```

Функция **longjmp()** возобновляет выполнение программы с места последнего вызова функции **setjmp()**. Эти две функции позволяют переключать поток управления с одной функции на другую. Обратите внимание на то, что для использования этой функции необходим заголовок **<csetjmp>**.

Функция **longjmp()** восстанавливает состояние стека, записанное в буфере *enfbuf* при предыдущем вызове функции **setjmp()**. В результате выполнение программы возобновляется с оператора, следующего за вызовом функции **setjmp()**. Иначе говоря, компьютеру "кажется", что поток управления никогда не покидал функцию, вызвавшую функцию **setjmp()**. (Образно выражаясь, функция **longjmp()** представляет собой некую машину времени, которая возвращает управление в предыдущую точку программы, как ни в чем ни бывало.)

Буфер *envbuf* имеет тип **jmp_buf**, определенный в заголовке **<csetjmp>**. Содержание буфера должно задаваться с помощью вызова функции **setjmp()** еще до вызова функции **longjmp()**.

Значение параметра *status* становится значением, возвращаемым функцией **setjmp()**. Оно используется для определения отправной точки, из которой выполняется нелокальный переход. Допускаются все значения, кроме нуля.

Чаще всего функция **longjmp()** используется для возврата из глубоко вложенного набора функций при возникновении ошибки.

Зависимая функция: **setjmp()**.

Функция mblen

```
#include <cstdlib>
int mblen(const char *str, size_t size);
```

Функция **mblen()** возвращает длину многобайтового символа (в байтах), на который ссылается указатель *str*. Функция проверяет лишь *size* первых символов. В случае ошибки возвращается число -1.

Если указатель *str* является нулевым, а многобайтовые символы закодированы в соответствии с текущей локализацией, функция **mblen()** возвращает ненулевое значение. В противном случае возвращается число 0.

Зависимые функции: **mbtowc()** и **wctomb()**.

Функция mbstowcs

```
#include <cstdlib>
size_t mbstowcs(wchar_t *out, const char *in, size_t size);
```

Функция **mbstowcs()** преобразует многобайтовую строку *in* в расширенную символьную строку и записывает результат в массив *out*. При этом в массив *out* записываются лишь *size* первых байтов.

Функция **mbstowcs()** возвращает количество преобразованных многобайтовых символов. Если возникает ошибка, функция возвращает число -1.

Зависимые функции: **wcstombs()** и **mbtowc()**.

Функция mbtowc

```
#include <cstdlib>
int mbtowc(wchar_t *out, const char *in, size_t size);
```

Функция **mbtowc()** преобразует многобайтовый символ, записанный в массиве *in*, в расширенный символ и записывает результат в массив *out*. При этом учитываются лишь *size* первых байтов.

Функция **mbtowc()** возвращает количество преобразованных многобайтовых символов. Если возникает ошибка, функция возвращает число -1.

Зависимые функции: **mblen()** и **wctomb()**.

Функция qsort

```
#include <cstdlib>
void qsort(void *buf, size_t num, size_t size,
           int (*compare) (const void *, const void *));
```

Функция **qsort()** упорядочивает массив *buf* с помощью алгоритма быстрой сортировки (разработанного Хоаром (Hoare)). Алгоритм быстрой сортировки является наилучшим универсальным алгоритмом сортировки. После выполнения функции массив оказывается упорядоченным. Количество элементов массива задается параметром *num*, а размер каждого элемента (в байтах) задается параметром *size*.

Для сравнения элементов массива используется функция *compare*. Объявление функции **compare()** должно выглядеть следующим образом.

```
int func_name(const void *arg1, const void *arg2);
```

Она должна возвращать значения, описанные в следующей таблице.

Сравнение	*Возвращаемое значение*
arg1 меньше *arg2*	Меньше нуля
arg1 равно *arg2*	Нуль
arg1 больше *arg2*	Больше нуля

Массив сортируется в порядке возрастания.

Зависимая функция: **bsearch()**.

Функция raise

```
#include <csignal>
int raise(int signal);
```

Функция **raise()** посылает выполняемой программе сигнал, определенный параметром *signal*. Если функция выполнена успешно, возвращается нуль, в противном случае возвращается ненулевое значение. Для применения функции **raise()** необходим заголовок **<csignal>**.

Стандарт языка C++ определяет следующие сигналы. Разумеется, каждый компилятор может добавлять свой набор дополнительных сигналов.

Макрос	Значение
SIGABRT	Аварийное завершение программы.
SIGFPE	Ошибка при выполнении операции над числами с плавающей точкой
SIGILL	Неверная команда.
SIGINT	Пользователь нажал комбинацию клавиш <CTRL+C>.
SIGSEGV	Неверный доступ к памяти.
SIFTERM	Прекратить выполнение программы.

Зависимая функция: **signal()**.

Функция rand

```
#include <cstdlib>
int rand(void);
```

Функция **rand()** генерирует последовательность псевдослучайных чисел Каждый раз при вызове функции **rand()** возвращается целое число из диапазона от нуля до **RAND_MAX**.

Зависимая функция: **srand()**.

Функция setjmp

```
#include <csetjmp>
int setjmp(jmp_buf envbuf);
```

Функция **setjmp()** сохраняет в буфере *envbuf* состояние стека в момент последнего вызова функции **longjmp()**. Для использования этой функции необходим заголовок **<csetjmp>**.

При вызове функция **setjmp()** возвращает число 0. Однако функция **longjmp()** передает функции **setjmp()** аргумент, который становится ее значением (всегда ненулевым) после вызова функции **longjmp()**.

Дополнительная информация содержится в разделе, посвященном функции **longjmp()**.

Зависимая функция: **longjmp()**.

Функция signal

```
#include <csignal>
void (*signal(int signal, void (*func)(int))) (int);
```

Функция **signal()** регистрирует обработчик сигнала, заданного параметром *signal*. Иначе говоря, когда программа получит сигнал, определенный параметром *signal*, будет вызвана функция, адресованная указателем *func*.

Значением указателя *func* является адрес обработчика сигнала или один из следующих макросов, определенных в заголовке **<csignal>**.

Макрос	Значение
SIG_DFL	Использовать стандартный обработчик ошибки.
SIG_IGN	Игнорировать сигнал.

Если используется адрес функции, при получении сигнала выполняется его обработка.

В случае успеха функция **signal()** возвращает адрес функции, ранее определенной для данного сигнала. При возникновении ошибки возвращается макрос **SIG_ERR**, определенный в заголовке **<csignal>**.

Зависимая функция: **raise()**.

Функция srand

```
#include <cstdlib>
void srand(unsigned seed);
```

Функция **srand()** используется для вычисления стартовой точки при генерации последовательности псевдослучайных чисел, возвращаемых функцией **rand()**.

Как правило, функцию **srand()** применяют для генерации нескольких разных последовательностей псевдослучайных чисел, задавая разные стартовые точки. Кроме того, с помощью функции **srand()** можно генерировать одну и ту же последовательность псевдослучайных чисел, вновь и вновь вызывая функцию с одним и тем же аргументом.

Зависимая функция: **rand()**.

Функция strtod

```
#include <cstdlib>
double strtod(const char *start, char **end);
```

Функция **strtod()** преобразует представление числа, хранящегося в строке *start*, в значение типа **double** и возвращает результат.

Функция **strtod()** работает следующим образом. Сначала из строки *start* выбрасываются все разделители. Затем считываются символы, образующие число. Если какой-либо из считанных символов не может быть элементом записи числа, процесс прекращается. Это относится к разделителям, знакам пунктуации (кроме точки) и символам, отличным от букв **E** и **e**. В заключение указатель *end* устанавливается на остаток исходной строки, если он существует. Это значит, что при вызове "1000.00 Hello" будет возвращено число 100.00, а указатель *end* будет установлен на пробел, предшествующий строке "Hello".

Если преобразование выполнить не удалось, функция возвращает 0. При переполнении функция **strtod()** возвращает константу **HUGE_VAL** или **-HUGE_VAL**, соответствующую положительному или отрицательному переполнению, а глобальной переменной **errno** присваивается значение **ERANGE**, означающее выход за пределы допустимых значений. В случае потери значимости возвращается нуль, а глобальной переменной **errno** снова присваивается значение **ERANGE**.

Зависимая функция: **atof()**.

Функция strtol

```
#include <cstdlib>
long strtol(const char *start, char **end, int radix);
```

Функция **strtol()** преобразует представление числа, хранящегося в строке *start*, в значение типа **long** и возвращает результат. Основание системы счисления, в которой представлено число, задается параметром *radix*. Если значение *radix* равно нулю, основание системы счисления определяется по правилам спецификации констант (т.е. десятичные числа должны начинаться с цифры, восьмеричные — с буквы **o**,

а шестнадцатеричные — с префикса 0x. — *Прим. ред.*). Кроме нуля параметр *radix* может принимать значения от 2 до 36.

Функция **strtol()** работает следующим образом. Сначала из строки *start* выбрасываются все разделители. Затем считываются символы, образующие число. Если какой-либо из считанных символов не может быть элементом записи значения типа **long**, процесс прекращается. Это относится к разделителям, знакам пунктуации и символам. В заключение указатель *end* устанавливается на остаток исходной строки, если он существует. Это значит, что при вызове "1000.00 Hello" будет возвращено число 100L, а указатель *end* будет установлен на пробел, предшествующий строке "Hello".

Если преобразование выполнить не удалось, функция возвращает число 0. При переполнении функция **strtol()** возвращает константу **LONG_MAX** или **LONG_MIN**, а глобальной переменной **errno** присваивается значение **ERANGE**, означающее выход за пределы допустимых значений. В случае потери значимости возвращается нуль, а глобальной переменной **errno** снова присваивает значение **ERANGE**.

Зависимая функция: **atol()**.

Функция strtoul

```
#include <cstdlib>
unsigned long strtoul(const char *start, char **end, int radix);
```

Функция **strtoul()** преобразует представление числа, хранящегося в строке *start*, в значение типа **unsigned long** и возвращает результат. Основание системы счисления, в которой представлено число, задается параметром *radix*. Если значение *radix* равно нулю, основание системы счисления определяется по правилам спецификации констант. Кроме нуля параметр *radix* может принимать значения от 2 до 36.

Функция **strtoul()** работает следующим образом. Сначала из строки *start* выбрасываются все разделители. Затем считываются символы, образующие число. Если какой-либо из считанных символов не может быть элементом записи значения типа **unsigned long**, процесс прекращается. Это относится к разделителям, знакам пунктуации и символам. В заключение указатель *end* устанавливается на остаток исходной строки, если он существует. Это значит, что при вызове "100.00 Hello" будет возвращено число 100L, а указатель *end* будет установлен на пробел, предшествующий строке "Hello".

Если преобразование выполнить не удалось, функция возвращает число 0. При переполнении функция **strtoul()** возвращает константу **ULONG_MAX** или **ULONG_MIN**, а глобальной переменной **errno** присваивает значение **ERANGE**, означающее выход за пределы допустимых значений. В случае потери значимости возвращается нуль, а глобальной переменной **errno** снова присваивается значение **ERANGE**.

Зависимая функция: **strtol()**.

Функция system

```
#include <cstdlib>
int system(const char *str);
```

Функция **system()** передает центральному процессору операционной системы команду, представленную в виде строки *str*.

Если функция **system()** применяется к нулевому указателю, она вернет ненулевое значение, если процессор доступен, и 0 — в противном случае. (Некоторые программы на языке C++ выполняются в специальных средах, не имеющих опера-

ционной системы и центрального процессора, поэтому вполне возможна ситуация, при которой процессор действительно не доступен.) Значение, возвращаемое функцией **system()**, зависит от конкретной реализации. Однако, как правило, если команда была успешно выполнена, функция возвращает нуль, в противном случае — ненулевое значение.

Зависимая функция: **exit()**.

Функции va_arg, va_start и va_end

```
#include <cstdarg>
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

Макросы **va_arg()**, **va_start()** и **va_end()** позволяют передавать функциям переменное количество аргументов. Наиболее ярким примером функции, имеющей переменное количество аргументов, является функция **printf**. Тип **va_list** определен в заголовке **<cstdarg>**.

Общая процедура создания функции, имеющей переменное количество аргументов, состоит из следующих этапов. Функция должна иметь как минимум один известный параметр, предшествующий переменному списку аргументов. Крайний справа известный параметр называется *last_parm*. Это имя используется в вызове функции **va_start()** в качестве второго параметра. Чтобы получить доступ к любому параметру, указанному в списке переменной длины, необходимо инициализировать указатель *argptr* с помощью макроса **va_start()**. Параметры возвращаются с помощью макроса **va_arg()**, вторым параметром которого является имя типа *type*. В заключение, когда все параметры считаны, перед возвращением из функции следует вызвать макрос **va_end()**, восстанавливающий состояние стека. Если этого не сделать, программа завершится аварийно.

Зависимая функция: **vprintf()**.

Функция wcstombs

```
#include <cstdlib>
size_t wcstombs(char *out, const wchar_t *in, size_t size);
```

Функция **wcstombs()** преобразует массив расширенных символов *in* в многобайтовый эквивалент и записывает результат в массив *out*. При этом учитываются лишь *size* первых байтов в строке *in*. В случае обнаружения признака конца строки процесс преобразования останавливается.

В случае успеха функция **wcstombs()** возвращает количество преобразованных байтов. Если возникает ошибка, функция возвращает число -1.

Зависимые функции: **wctomb()** и **mbstowcs()**.

Функция wctomb

```
#include <cstdlib>
int wctombs(char *out, wchar_t *in);
```

Функция **wctomb()** преобразует расширенный символ *in* в многобайтовый эквивалент и записывает результат в массив *out*, длина которого не должна превышать значение **MB_CUR_MAX**.

В случае успеха функция **wctomb()** возвращает количество преобразованных байтов, содержащихся в многобайтовом символе. Если возникает ошибка, функция возвращает число -1.

Если указатель *out* является нулевым, функция **wctomb()** возвращает ненулевое значение, если многобайтовый символ закодирован в соответствии с текущей локализацией. В противном случае возвращается число 0.

Зависимые функции: **wcstombs()** и **mbtowc()**.

Полный справочник по C++

Глава 31

Функции обработки расширенных символов

В 1995 году в стандарт языка С вошли функции обработки расширенных символов, которые впоследствии стали частью стандарта языка C++. Эти функции оперируют символами типа **wchar_t** длиной 16 бит. Большинство этих функций имеют аналоги, работающие с символами типа **char**. Например, функция **iswspace()** является вариантом функции **isspace()**. Как правило, имена функций обработки расширенных символов отличаются от имен соответствующих символьных функций одной буквой — "**w**".

Функции обработки расширенных символов используют два заголовка: **<cwchar>** и **<cwctype>**. Программы на языке С используют заголовочные файлы **wchar.h** и **wctype.h**.

В заголовке **<cwctype>** определены типы **wint_t**, **wctrans_t** и **wctype_t**. Многие из функций обработки расширенных символов получают в качестве параметра расширенный символ, имеющий тип **wint_t**. Этот тип является аналогом типа **int**, применяемого в символьных функциях. Типы **wctrans_t** и **wctype_t** используются соответственно для преобразования символов и их классификации. Расширенный признак конца файла определен константой **WEOF**.

Кроме типа **wint_t** в заголовке **<cwchar>** определены типы **wchar_t**, **size_t** и **mbstate_t**. Тип **wchar_t** создает расширенный символ, а тип **size_t** задает тип значения, возвращаемого оператором **sizeof**. Тип **mbstate_t** описывает объект, в котором хранится информация о преобразовании многобайтового символа в расширенный. В заголовке **<cwchar>** определены также макросы **NULL**, **WEOF**, **WCHAR_MAX** и **WCHAR_MIN**. Последние два макроса определяют максимальное и минимальное значения, которые могут храниться в объекте типа **wchar_t**.

Несмотря на то что функции обработки расширенных символов представлены в стандартной библиотеке достаточно широко, они применяются редко. Одна из причин этого заключается в том, что система ввода-вывода языка C++ и библиотека классов обеспечивают работу с обычными и расширенными символами с помощью шаблонных классов. Кроме того, интерес к программам, использующим расширенные символы, оказался меньше, чем ожидалось. Разумеется, со временем эта ситуация может измениться.

Поскольку многие функции обработки расширенных символов аналогичны символьным и довольно редко применяются большинством программистов, наш обзор будет кратким.

Функции классификации расширенных символов

В заголовке **<cwctype>** содержатся прототипы функций классификации расширенных символов. Эти функции классифицируют расширенные символы или преобразуют строчные символы в прописные, и наоборот. Эти функции перечислены в табл. 31.1 вместе с их символьными эквивалентами, описанными в главе 26.

Таблица 31.1. Функции классификации расширенных символов

Функция	*Символьный эквивалент*
`int iswalnum(wint_t ch)`	`isalnum()`
`int iswalpha(wint_t ch)`	`isalpha()`
`int iswcntrl(wint_t ch)`	`iscntrl()`
`int iswdigit(wint_t ch)`	`isdigit()`
`int iswgraph(wint_t ch)`	`isgraph()`

Окончание табл. 31.1

Функция	Символьный эквивалент
`int iswlower(wint_t ch)`	`islower()`
`int iswprint(wint_t ch)`	`isprint()`
`int iswpunct(wint_t ch)`	`ispunct()`
`int iswspace(wint_t ch)`	`isspace()`
`int iswupper(wint_t ch)`	`isupper()`
`int iswxdigit(wint_t ch)`	`isxdigit()`
`wint_t tolower(wint_t ch)`	`tolower()`
`wint_t toupper(wint_t ch)`	`toupper()`

Кроме функций, перечисленных в табл. 31.1, в заголовке **<cwctype>** определены функции, предоставляющие расширяемые символы классификации символов.

```
wctype_t wctype(const char *attr);
int iswctype(wint_t ch, wctype_t attr_ob);
```

Функция **wctype()** возвращает значение, которое можно передать функции **iswctype()** в качестве параметра *attr_ob*. Строка *attr* задает свойства, которыми должен обладать символ. Значение параметра *attr_ob* позволяет определить, обладает ли символ *ch* указанными свойствами. В случае положительного ответа функция **iswctype()** возвращает ненулевое значение, иначе она возвращает нуль. Свойства, перечисленные ниже, поддерживаются всеми без исключения средами выполнения программы.

alnum	**alpha**	**cntrl**	**digit**
graph	**lower**	**print**	**punct**
space	**upper**	**xdigit**	

Следующая программа демонстрирует применение функций **wctype()** и **iswctype()**.

```
#include <iostream>
#include <cwctype>
using namespace std;

int main()
{
  wctype_t x;

  x = wctype("space");

  if(iswctype(L' ', x))
    cout << "Пробел.\n";

  return 0;
}
```

Программа выводит на экран слово "Пробел.".

Функции **wctrans()** и **towctrans()** также определены в заголовке **<cwctype>**.

```
wctrans_t wctrans(const char *mapping);
wint_t towctrans(wint_t ch, wctrans_t *mapping_ob);
```

Функция **wctrans()** возвращает значение, которое можно передать функции **towctrans()** в качестве параметра *mapping_ob*. Строка *mapping* задает преобразование одного символа в другой. Это значение позволяет функции **iswctrans()** выполнить преобразование символа *ch* и вернуть его результат. Свойства, перечисленные ниже, поддерживаются всеми без исключения средами выполнения программы.

tolower **toupper**

Рассмотрим короткий пример, иллюстрирующий применение функций **wctrans()** и **towctrans()**.

```
#include <iostream>
#include <cwctype>
using namespace std;

int main()
{
  wctrans_t x;

  x = wctrans("tolower");

  wchar_t ch = towctrans(L'W', x);
  cout << (char) ch;

  return 0;
}
```

Функции ввода-вывода расширенных символов

В главе 25 уже упоминались некоторые функции ввода-вывода расширенных символов. В табл. 31.2 эти функции определены в заголовке **<cwchar>**. Обратите внимание на то, что функции **swprintf()** и **vswprintf()** нуждаются в дополнительном параметре, и этим они отличаются от своих символьных эквивалентов.

Таблица 31.2. Функции классификации расширенных символов

Функция	*Символьный эквивалент*
`wint_t fgetwc(FILE *stream)`	`fgetc()`
`wint_t *fgetws (wchar_t *str, int num, FILE *stream)`	`fgets()`
`wint_t fputwc (wchar_t *ch, FILE *stream)`	`fputc()`
`int fputws( const wchar_t *str, FILE *stream)`	`fputs()`
`int fwprintf (FILE *stream, const wchar_t fmt, ...)`	`fprintf()`
`int fwscanf(FILE *stream, const wchar_t fmt, ...)`	`fscanf()`
`wint_t getwc(FILE *stream)`	`getc()`
`wint_t getwchar()`	`getchar()`
`wint_t putwc (wchar_t ch, FILE *stream)`	`putc()`
`wint_t putwchar(wchar_t ch)`	`putchar()`

Окончание табл. 31.2

Функция	Символьный эквивалент
`int swprintf(wchar_t *str, size_t num, const wchar_t *fmt, ...)`	`sprintf()` Обратите внимание на дополнительный параметр *num*, ограничивающий количество символов, записываемых в строку *str*
`int swscanf(const wchar_t *str, const wchar_t *fmt, ...)`	`sscanf()`
`wint_t ungetwc (wint_t ch, FILE *stream)`	`ungetc()`
`int vfwprintf(FILE *stream, const wchar_t fmt, va_list arg)`	`vfprintf()`
`int vswprintf(wchar_t *str, size_t num, const wchar_t *fmt, va_list arg)`	`vsprintf()`
`int vwprintf (const wchar_t *fmt, va_list arg)`	Обратите внимание на дополнительный параметр *num*, ограничивающий количество символов, записываемых в строку *str*
`int vwprintf (const wchar_t *fmt, va_list arg)`	`vprintf()`
`int wprintf(const wchar_t *fmt, ...)`	`printf()`
`int wscanf(const wchar_t *fmt, ...)`	`scanf()`

Кроме функций, перечисленных в таблице, в языке C++ есть еще одна функция ввода-вывода расширенных символов.

```
int fwide(FILE *stream, int how);
```

Если параметр *how* положителен, функция **fwide()** создает поток расширенных символов *stream*. При отрицательном параметре *how* функция **fwide()** создает поток обычных символов. Если параметр *how* равен нулю, поток не изменяется. Поток, ориентированный на обычные или расширенные символы, также не будет изменен. Если поток использует расширенные символы, функция возвращает положительное значение. Если поток работает с обычными символами, возвращается отрицательное число. Если ориентация потока еще не установлена, функция возвращает нуль. Ориентация потока определяется при его первом применении.

Функции обработки строк, состоящих из расширенных символов

Аналоги строковых функций (см. главу 26), манипулирующие строками, содержащими расширенные символы, приведены в табл. 31.3. Для их применения необходим заголовок <cwchar>. Обратите внимание на то, что функции **wcstok()** необходим дополнительный параметр, который не используется ее символьным эквивалентом.

Таблица 31.3. Функции обработки строк расширенных символов

Функция	Символьный эквивалент
`wchar_t *wcscat(wchar_t *str1, const wchar_t *str2)`	`strcat()`
`wchar_t *wcschr(const wchar_t *str, wchar_t ch)`	`strchr()`
`int wcscmp(const wchar_t *str1, const wchar_t *str2)`	`strcmp()`

Окончание табл. 31.3

Функция	Символьный эквивалент
`int wcscoll(const wchar_t *str1, const wchar_t *str2)`	`strcoll()`
`size_t wcscspn(const wchar_t *str1, const wchar_t *str2)`	`strcspn()`
`wchar_t *wcscpy(wchar_t *str1, const wchar_t *str2)`	`strcpy()`
`size_t wcslen(const wchar_t *str)`	`strlen()`
`wchar_t *wcsncpy(wchar_t *str1, const wchar_t *str2, size_t num)`	`strncpy()`
`wchar_t *wcsncat(wchar_t *str1, const wchar_t *str2, size_t num)`	`strncat()`
`int wcsncmp(const wchar_t *str1, const wchar_t *str2, size_t num)`	`strncmp()`
`wchar_t *wcspbrk(const wchar_t *str1, const wchar_t *str2)`	`strpbrk()`
`wchar_t *wcsrchr(const wchar_t *str, wchar_t ch)`	`strrchr()`
`size_t wcsspn(const wchar_t *str1, const wchar_t *str2)`	`strspn()`
`wchar_t *wcstok (wchar_t *str1, const wchar_t *str2, wchar_t **endptr)`	`strtok()` Указатель *endptr* содержит информацию, необходимую для продолжения разбиения строки на лексемы
`wchar_t *wcsstr(const wchar_t *str1, const wchar_t *str2)`	`strstr()`
`size_t wcsxfrm(wchar_t *str1, const wchar_t *str2, size_t num)`	`strxfrm()`

Функции преобразования строк, состоящих из расширенных символов

Функции, перечисленные в табл. 31.4, являются аналогами стандартных функций преобразования чисел и времени. Они используют заголовок **<cwchar>**.

Таблица 31.4. Функции преобразования расширенных символов

Функция	Символьный эквивалент
`size_t wcsftime(wchar_t *str1, size_t max, const wchar_t *fmt, const struct tm *ptr)`	`strftime()`
`double wcstod(const wchar_t *start, wchar_t **end)`	`strtod()`
`long wcstol(const wchar_t *start, wchar_t **end, int radix)`	`strtol()`
`unsigned long wcstoul(const wchar_t *start, wchar_t **end, int radix)`	`strtoul()`

Функции обработки массивов расширенных символов

Для стандартных функций, обрабатывающих массивы символов, например **memcpy()**, также существуют расширенные аналоги. Они приведены в табл. 31.5. Эти функции используют заголовок **<cwchar>**.

Таблица 31.5. Функции обработки массивов расширенных символов

Функция	Символьный эквивалент
`wchar_t *wmemhr(const wchar_t *str, wchar_t ch, size_t num)`	`memchr()`
`int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t num)`	`memcmp()`
`wchar_t *wmemcpy(wchar_t *str1, const wchar_t *str2, size_t num)`	`memcpy()`
`wchar_t *wmemmove(wchar_t *str1, const wchar_t *str2, size_t num)`	`memmove()`
`wchar_t *wmemset(wchar_t *str, wchar_t ch, size_t num)`	`memset()`

Функции преобразования многобайтовых и расширенных символов

В стандартной библиотеке языка C++ определены разнообразные функции, предназначенные для преобразования многобайтовых и расширенных символов. Эти функции, перечисленные в табл. 31.6, используют заголовок **<cwchar>**. Многие из них являются *возобновляемыми* (restartable) версиями обычных функций, предназначенных для преобразования многобайтовых символов. Возобновляемые версии используют информацию, передаваемую с помощью параметра типа **mbstate_t**. Если этот параметр равен нулю, функция создаст свой собственный объект типа **mbstate_t**.

Таблица 31.6. Функции преобразования многобайтовых и расширенных символов

Функция	Символьный эквивалент
`wint_t btowc(int ch)`	Преобразует символ *ch* в его расширенный эквивалент и возвращает результат. В случае ошибки возвращает константу **WEOF** или многобайтовый символ, если символ *ch* не является однобайтовым
`size_t mbrlen(const char *str, size_t num, mbstate_t *state)`	Возобновляемая версия функции **mblen()**, описанная параметром *state*. Возвращает положительное число, равное количеству байтов в следующем многобайтовом символе. Если следующий символ является нулевым, возвращается нуль. При ошибке возвращается отрицательное значение
`size_t mbrtowc(wchar_t *out, const char* in, size_t num, mbstate_t *state)`	Возобновляемая версия функции **mbtowc()**, описанная параметром *state*. Возвращает положительное число, равное количеству байтов в следующем многобайтовом символе. Если следующий символ является нулевым, возвращается нуль. При ошибке переменной **errno** присваивается макрос **EILSEQ**
`int mbsinit(const mbstate_t *state)`	Если параметр *state* представляет исходное состояние преобразования, функция возвращает значение **true**
`size_t mbsrtowcs(wchar_t *out, const char **in, size_t num, mbstate_t *state)`	Возобновляемая версия функции **mbstowcs()**, описанная параметром *state*. Отличается от функции **mbrstowcs()** тем, что параметр *in* является неявным указателем на исходный массив. При ошибке переменной **errno** присваивается макрос **EILSEQ**

Окончание табл. 31.6

Функция	*Символьный эквивалент*
`size_t wcrtomb(char *`*out*`, wchar_t` *ch*`, mbstate_t *`*state*`)`	Возобновляемая версия функции `wctomb()`, описанная параметром *state*. При ошибке переменной `errno` присваивается макрос `EILSEQ`
`size_t wcsrtombs(char *`*out*`, const wchar_t **`*in*`, size_t` *num*`, mbstate_t *`*state*`)`	Возобновляемая версия функции `wcstombs()`, описанная параметром *state*. Отличается от функции `wcstombs()` тем, что параметр *in* является неявным указателем на исходный массив. При ошибке переменной `errno` присваивается константа `EILSEQ`
`int wctob(wint_t` *ch*`)`	Преобразует символ *ch* в его однобайтовый эквивалент. При возникновении ошибки возвращается константа `EOF`

Полный справочник по C++

Часть IV

Библиотека стандартных классов

Стандарт языка C++ предусматривает большое количество классов, обеспечивающих выполнение широкого круга операций, включая ввод-вывод, обработку строк и математические вычисления. Библиотека классов является дополнением библиотеки функций, описанных в части III. Библиотека классов представляет собой весьма важную часть языка C++ и во многом определяет его характер. Несмотря на ее большой размер, освоить библиотеку классов достаточно легко, поскольку она основана на объектно-ориентированных принципах.

Полный справочник по C++

Глава 32

Стандартные классы ввода-вывода

В главе описывается библиотека стандартных классов ввода-вывода. Как указывалось в части II, в настоящее время язык C++ использует два варианта библиотеки ввода-вывода. Первая библиотека представляет собой устаревший набор функций, унаследованных от языка C и не описанных в стандарте языка C++. Вторая библиотека содержит классы, образующие современную шаблонную систему ввода-вывода, предусмотренную стандартом языка C++. Поскольку современная библиотека ввода-вывода является расширением старой, в этой главе мы сосредоточимся на изучении новых средств.

На заметку *Обзор средств ввода-вывода содержится в главах 20 и 21.*

Классы ввода-вывода

Стандартная система ввода-вывода языка C++ состоит из довольно сложных шаблонных классов, перечисленных ниже.

Класс	*Предназначение*
`basic_ios`	Универсальные операции ввода-вывода
`basic_streambuf`	Низкоуровневые операции ввода-вывода
`basic_istream`	Операции ввода
`basic_ostream`	Операции вывода
`basic_iostream`	Операции ввода-вывода
`basic_filebuf`	Низкоуровневые операции файлового ввода-вывода
`basic_ifstream`	Операции файлового ввода
`basic_ofstream`	Операции файлового вывода
`basic_fstream`	Операции файлового ввода-вывода
`basic_stringbuf`	Низкоуровневые операции строкового ввода-вывода
`basic_istringstream`	Операции строкового ввода
`basic_ostringstream`	Операции строкового вывода
`basic_stringstream`	Операции строкового ввода-вывода

В иерархию классов ввода-вывода входит также нешаблонный класс **`ios_base`**. Он содержит определения различных элементов системы ввода-вывода.

Система ввода-вывода состоит из двух связанных между собой, но разных иерархий шаблонных классов. В основе первой иерархии лежит класс низкоуровневого ввода-вывода **`basic_streambuf`**. Этот класс, обеспечивающий поддержку всей системы, предназначен для выполнения основных низкоуровневых операций ввода-вывода. Классы **`basic_filebuf`** и **`basic_stringbuf`** являются производными от класса **`basic_streambuf`**. В обычных приложениях нет необходимости непосредственно обращаться к классу **`basic_streambuf`** и его подклассам.

Для ввода-вывода в программах обычно используется иерархия классов, основанная на классе **`basic_ios`**. Этот класс обеспечивает высокоуровневые операции, предусматривающие форматирование, проверку ошибок и сохранение информации о состоянии потока ввода-вывода. Класс **`basic_ios`** является базовым по отношению ко многим производным классам, к которым, в частности, относятся классы **`basic_istream`**, **`ba-`**

sic_ostream и **basic_iostream**. Эти классы используются соответственно для создания потоков ввода, вывода и ввода-вывода. В частности, классы **basic_ifstream** и **basic_istringstream** являются производными от класса **basic_istream**, классы **basic_ofstream** и **basic_ostringstream** — от класса **basic_ostream**, а классы **basic_fstream** и **basic_stringstream** — от класса **basic_iostream**. Класс **basic_ios** является наследником класса **ios_base**. Таким образом, любой класс, производный от класса **basic_ios**, имеет доступ к членам класса **ios_base**.

Шаблонными параметрами классов ввода-вывода являются тип и атрибуты символов. Рассмотрим в качестве примера шаблонную спецификацию класса **basic_ios**.

```
template <class CharType, class Attr = char_traits<CharType> >
   class basic_ios:public ios_base
```

Здесь параметр **CharType** задает тип символа (т.е. **char** или **wchar_t**), а параметр **Attr** указывает тип, описывающий его атрибуты. Обобщенный тип **char_traits** представляет собой служебный класс, в котором определены атрибуты символа.

Как указано в главе 20, библиотека ввода-вывода создает две специализации шаблонных иерархий классов: одну для 8-битовых символов, а другую — для расширенных. Таблица соответствий между именами шаблонных классов, а также их версиями для обычных и расширенных символов приведена ниже.

Шаблонный класс	*Класс для символов*	*Класс для расширенных символов*
basic_ios	**ios**	**wios**
basic_istream	**istream**	**wistream**
basic_ostream	**ostream**	**wostream**
basic_iostream	**iostream**	**wiostream**
basic_ifstream	**ifstream**	**wifstream**
basic_ofstream	**ofstream**	**wofstream**
basic_fstream	**fstream**	**wfstream**
basic_istringstream	**istringstream**	**wistringstream**
basic_ostringstream	**ostringstream**	**wostringstream**
basic_stringstream	**stringstream**	**wstringstream**
basic_streambuf	**streambuf**	**wstreambuf**
basic_filebuf	**filebuf**	**wfilebuf**
basic_stringbuf	**stringbuf**	**wstringbuf**

Поскольку подавляющее большинство программистов используют в своих программах символьный ввод-вывод, именно эти классы мы будем применять в данной главе. Таким образом, говоря о классах ввода-вывода, мы будем указывать имена их версий, предназначенных для работы с символами, а не их внутренние шаблонные имена. Например, мы упоминаем классы **ios**, а не **basic_ios**; **istream**, а не **basic_istream** и **fstream**, а не **basic_fstream**. Следует помнить, что наряду с этими версиями существуют варианты классов, предназначенные для работы с расширенными символами, и работают они совершенно так же.

Заголовки ввода-вывода

Стандартная система ввода-вывода использует несколько заголовков.

Заголовок	Предназначение
`<fstream>`	Файловый ввод-вывод
`<iomanip>`	Параметризованные манипуляторы ввода-вывода
`<ios>`	Основные операции ввода-вывода
`<iosfwd>`	Неполные объявления, используемые системой ввода-вывода
`<iostream>`	Общие операции ввода-вывода
`<istream>`	Основные операции ввода
`<ostream>`	Основные операции вывода
`<sstream>`	Строковый вывод
`<streambuf>`	Низкоуровневые операции ввода-вывода

Некоторые из этих заголовков используются внутри системы ввода-вывода и пользователям не нужны. Как правило, в прикладных программах применяются лишь три заголовка: `<iostream>`, `<fstream>`, `<sstream>` и `<iomanip>`.

Флаги форматирования и манипуляторы ввода-вывода

Каждый поток связан с определенным набором флагов форматирования, управляющих внешним представлением информации. Класс `ios_base` определяет перечисление битовых масок под названием `fmtflags`, в котором задаются значения следующих флагов.

`adjustfield`	`basefield`	`boolalpha`	`dec`
`fixed`	`floatfield`	`hex`	`internal`
`left`	`oct`	`right`	`scientific`
`showbase`	`showpoint`	`showpos`	`skipws`
`unitbuf`	`uppercase`		

Эти значения используются для установки и сброса флагов форматирования с помощью функций, таких как `setf()` и `unsetf()`. Подробное описание этих флагов дано в главе 20.

Кроме непосредственной установки и сброса флагов форматирования параметры формата можно изменять с помощью специальных функций, называемых *манипуляторами*. Эти функции можно вставлять прямо в выражения ввода-вывода. Стандартные манипуляторы перечислены ниже.

Манипулятор	Предназначение	Ввод-вывод
`boolalpha`	Устанавливает флаг `boolalpha`.	Ввод-вывод
`dec`	Устанавливает флаг `dec`.	Ввод-вывод
`endl`	Выводит символ перехода на новую строку и очищает поток.	Вывод
`ends`	Выводит нулевой символ, служащий признаком конца строки.	Вывод
`fixed`	Устанавливает флаг `fixed`.	Вывод
`flush`	Очищает поток.	Вывод
`hex`	Устанавливает флаг `hex`.	Ввод-вывод
`internal`	Устанавливает флаг `internal`.	Выеод
`left`	Устанавливает флаг `left`.	Вывод
`noboolalpha`	Сбрасывает флаг `boolalpl a`.	Ввод-вывод
`noshowbase`	Сбрасывает флаг `showbase`.	Вывод

Окончание табл.

Манипулятор	*Предназначение*	*Ввод-вывод*
noshowpoint	Сбрасывает флаг **showpoint**.	Вывод
noshowpos	Сбрасывает флаг **showpos**.	Вывод
noskipws	Сбрасывает флаг **skipws**.	Ввод
nounitbuf	Сбрасывает флаг **unitbuf**.	Вывод
nouppercase	Сбрасывает флаг **uppercase**.	Вывод
oct	Устанавливает флаг **oct**.	Ввод-вывод
resetiosflags (fmtflags *f*)	Сбрасывает все флаги, указанные в перечислении *f*.	Ввод-вывод
right	Устанавливает флаг **right**.	Вывод
scientific	Устанавливает флаг **scientific**.	Вывод
setbase(int *base*)	Устанавливает основание системы счисления, указанное параметром *base*.	Ввод-вывод
setfill(int *ch*)	Устанавливает символ-заполнитель, заданный параметром *ch*.	Вывод
setiosflags (fmtflags *f*)	Устанавливает все флаги, указанные в перечислении *f*.	Ввод-вывод
setprecision (int *p*)	Устанавливает точность представления числа, т.е. количество знаков после десятичной точки.	Вывод
setw(int *w*)	Устанавлиеает ширину поля вывода, указанную параметром *w*.	Вывод
showbase	Устанавливает флаг **showbase**.	Вывод
showpoint	Устанавливает флаг **showpoint**.	Вывод
showpos	Устанавливает флаг **showpos**.	Вывод
skipws	Устанавливает флаг **skipws**.	Вывод
unitbuf	Устанавливает флаг **unutbuf**.	Вывод
uppercase	Устанавливает флаг **uppercase**.	Вывод
ws	Игнорирует ведущие разделители.	Ввод

Для применения манипуляторов с параметрами необходимо включить в программу заголовок **<iomanip>**.

Некоторые типы данных

Кроме описанного выше типа **fmtflags** стандартная система ввода-вывода определяет еще несколько типов данных.

Типы streamsize и streamoff

Объект типа **streamsize** содержит наибольшее количество байтов, которые можно передавать при выполнении любой из операций ввода-вывода. Обычно этот тип является разновидностью типа **int**. Объект типа **streamoff** содержит смещение относительно начала потока. Этот тип определен в заголовке **<ios>**, автоматически включаемом системой ввода-вывода.

Типы streampos и wstreampos

Объект типа **streampos** хранит значение, задающее позицию внутри потока **char**. Объект типа **wstreampos** хранит значение, задающее позицию внутри потока **wchar_t**. Оба типа определены в заголовке **<iosfwd>**, автоматически включаемом системой ввода-вывода.

Типы pos_type и off_type

Объекты типов **pos_type** и **off_type** (как правило, целочисленных) хранят значение, задающее позицию и смещение внутри потока. Эти типы определены в классе **ios** (и других классах) и, по существу, совпадают с типами **streamoff** и **streampos** (или их эквивалентами для расширенных символов).

Тип openmode

Тип **openmode** определен в классе **ios_base** и описывает режим открытия файла. Объект этого класса может принимать следующие значения.

app	Добавить запись в конец файла.
ate	Установить файловый курсор на конец файла.
binary	Открыть файл в бинарном режиме.
in	Открыть файл для ввода.
out	Открыть файл для вывода.
trunc	Стереть существующий файл.

Несколько значений можно комбинировать с помощью логической операции "ИЛИ".

Тип iostate

Объект типа **iostate** описывает текущее состояние потока ввода-вывода. Этот объект представляет собой перечисление, определенное в классе **ios_base**, который содержит следующие члены.

Имя	*Предназначение*
goodbit	Ошибок не обнаружено.
eofbit	Обнаружен конец файла.
failbit	Произошла поправимая ошибка.
badbit	Произошла непоправимая ошибка.

Тип seekdir

Тип **seekdir** описывает допустимые операции произвольного доступа. Он определен в классе **ios_base**. Объекты класса **seekdir** принимают следующие значения.

beg	Начало файла
cur	Текущее положение
end	Конец файла

Класс failure

В классе **ios_base** определен тип исключительной ситуации **failure**. Являясь производным от класса **exception** (стандартный класс исключительных ситуаций), он служит базовым классом для типов исключительных ситуаций, генерируемых системой ввода-вывода. Класс **failure** содержит следующий конструктор.

```
explicit failure(const string &str);
```

Параметр *str* представляет собой сообщение, описывающее ошибку. Это сообщение выдается объектом класса **failure** при вызове функции **what()**.

```
virtual const char *what() const throw();
```

Перегрузка операторов "<<" и ">>"

Операторы "<<" и/или ">>" для всех встроенных типов перегружены в следующих классах.

basic_istream
basic_ostream
basic_iostream

Эти операторы наследуются всеми классами, производными от этих классов.

Универсальные функции ввода-вывода

Оставшаяся часть главы посвящена универсальным функциям ввода-вывода, предусмотренным стандартом языка C++. Как известно, система ввода-вывода языка C++ основана на запутанной иерархии шаблонных классов. Многие из функций — членов низкоуровневых классов вообще не используются в прикладных программах. По этой причине мы их не описываем.

Функция bad

```
#include <iostream>
bool bad() const;
```

Функция **bad()** является членом класса **ios**.

В случае непоправимой ошибки, возникшей в потоке, функция **bad()** возвращает значение **true**, в противном случае возвращается значение **false**.

Зависимая функция: **good()**.

Функция clear

```
#include <iostream>
void clear(iostate flags = goodbit);
```

Функция **clear()** является членом класса **ios**.

Функция **clear()** сбрасывает флаг состояния потока. Если параметр *flags* равен значению **goodbit** (заданному по умолчанию), все флаги ошибок сбрасываются (приравниваются нулю). В противном случае флагу состояния присваивается значение, заданное параметром *flags*.

Зависимая функция: **rdstate()**.

Функция eof

```
#include <iostream>
bool eof() const;
```

Функция **eof()** является членом класса **ios**.

Если при вводе обнаружен конец потока ввода, функция **eof()** возвращает значение **true**, в противном случае возвращается значение **false**.

Зависимые функции: **bad()**, **fail()**, **good()**, **rdstate()** и **clear()**.

Функция exceptions

```
#include <iostream>
iostate exceptions() const;
void exceptions(iostate flags);
```

Функция **exceptions()** является членом класса **ios**.

Первый вариант возвращает объект класса **iostate**, позволяющий идентифицировать флаги, вызвавшие исключительную ситуацию. Второй вариант устанавливает эти флаги.

Зависимая функция: **rdstate()**.

Функция fail

```
#include <iostream>
bool fail() const;
```

Функция **fail()** является членом класса **ios**.

Если в потоке возникла ошибка, функция **fail()** возвращает значение **true**, в противном случае возвращается значение **false**.

Зависимые функции: **bad()**, **good()**, **eof()**, **rdstate()** и **clear()**.

Функция fill

```
#include <iostream>
char fill() const;
char fill(char ch);
```

Функция **fiil()** является членом класса **ios**.

По умолчанию при необходимости поле заполняется пробелами. Однако программист может задать свой символ-заполнитель, вызвав функцию **fill()** и указав желательное значение параметра *ch*. Функция возвращает старый символ-заполнитель.

Чтобы получить текущий символ-заполнитель, необходимо использовать первую версию функции **fill()**.

Зависимые функции: **precision()** и **width()**.

Функция flags

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

Функция **flags()** является членом класса **ios**, унаследованным от класса **ios_base**.

Первая версия функции **flags()** просто возвращает текущие флаги форматирования, установленные для соответствующего потока.

Вторая версия устанавливает флаги форматирования, заданные параметром *f*. В этом случае биты, образующие маску *f*, присваиваются флагам форматирования, связанным с соответствующим потоком.

Зависимые функции: **unsetf()** и **setf()**.

Функция flush

```
#include <iostream>
ostream &flush();
```

Функция **flush()** является членом класса **ostream**

Она принудительно записывает содержимое буфера, связанного с потоком, на устройство вывода. Функция возвращает ссылку на соответствующий поток.

Зависимые функции: **put()** и **write()**.

Функции fstream, ifstream и ofstream

```
#include <fstream>
fstream();
explicit fstream(const char *filename,
                 ios::openmode mode = ios::in | ios::out);
ifstream();
explicit ifstream(const char *filename,
                  ios::openmode mode = ios::in);

ofstream();
explicit ofstream(const char *filename,
                  ios::openmode mode = ios::out);
```

Функции **fstream()**, **ifstream()** и **ofstream()** являются конструкторами классов **fstream**, **ifstream** и **ofstream** соответственно.

Версии функций **fstream()**, **ifstream()** и **ofstream()**, не имеющие параметров, создают поток, не связанный ни с одним файлом. Впоследствии этот поток можно связать с файлом, вызвав функцию **open()**.

Функции **fstream()**, **ifstream()** и **ofstream()**, первым параметром которых является имя файла, в прикладных программах используются чаще других. Кроме того, обычно не требуется открывать файлы с помощью функции **open()**, поскольку конструкторы **fstream()**, **ifstream()** и **ofstream()** автоматически открывают файл при создании потока. Эти конструкторы и функция **open()** имеют одинаковые параметры, а также их значения, заданные по умолчанию. (См. раздел, посвященный функции **open()**.) Например, чаще всего файл открывается следующим образом.

```
ifstream mystream("myfile");
```

Если по какой-то причине файл открыть невозможно, значение соответствующего потока становится равным **false**. Следовательно, независимо от того, открываете вы файл с помощью конструктора или функции **open()**, необходимо проверить значение потока и убедиться, что файл действительно открыт.

Зависимые функции: **close()** и **open()**.

Функция gcount

```
#include <iostream>
streamsize gcount() const;
```

Функция **gcount()** является членом класса **istream**. Она возвращает количество символов, считанных при выполнении последней операции ввода.

Зависимые функции: **get()**, **getline()** и **read()**.

Функция get

```
#include <iostream>
int get();
istream &get(char &ch);
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
```

```
istream &get(streambuf &buf);
istream &get(streambuf &buf, char delim);
```

Функция **get()** является членом класса **istream**.

Как правило, функция **get()** считывает символы из входного потока. Версия функции **get()** без параметров считывает единственный символ из соответствующего потока и возвращает его в качестве результата.

Версия **get(char &ch)** считывает символ из потока и записывает его в переменную *ch*. Эта функция возвращает ссылку на поток.

Функция **get(char *buf, streamsize num)** считывает символы в массив *buf*, пока либо не будет введен *num*–1 символ, либо встретится символ перехода на новую строку, либо будет обнаружен конец файла. Если во входном потоке обнаружен символ перехода на новую строку, он *не* считывается, а остается в потоке до следующей операции чтения. Функция автоматически добавляет в конец массива нулевой символ и возвращает ссылку на поток.

Функция **get(char *buf, streamsize num, char delim)** считывает символы из массива *buf*, пока либо не будет введен *num*–1 символ, либо встретится разделитель *delim*, либо будет обнаружен конец файла. Если во входном потоке встречается символ *delim*, он *не* считывается, а остается в потоке до следующей операции чтения. Функция автоматически добавляет в конец массива нулевой символ и возвращает ссылку на поток.

Функция **get(streambuf &buf)** считывает символы из входного потока в объект класса **streambuf**. Символы считываются, пока не встретится символ перехода на новую строку или будет обнаружен конец файла. Если во входном потоке встречается символ перехода на новую строку, он *не* считывается. Функция возвращает ссылку на поток.

Функция **get(streambuf &buf, char delim)** считывает символы из входного потока в объект класса **streambuf**. Символы считываются, пока не встретится разделитель ***delim*** или будет обнаружен конец файла. Если во входном потоке встречается символ ***delim***, он *не* считывается. Функция возвращает ссылку на поток.

Зависимые функции: **put()**, **read()** и **getline()**.

Функция getline

```
#include <iostream>
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

Функция **getline()** является членом класса **istream**.

Функция **getline(char *buf, streamsize num)** считывает символы в массив *buf*, пока либо не будет введен *num*–1 символ, либо встретится символ перехода на новую строку, либо будет обнаружен конец файла. Если во входном потоке встречается символ перехода на новую строку, он считывается, но не записывается в массив *buf*. Функция автоматически добавляет в конец массива нулевой символ и возвращает ссылку на поток.

Функция **get(char *buf, streamsize num, char delim)** считывает символы из массива *buf*, пока либо не будет введен *num*–1 символ, либо встретится разделитель *delim*, либо будет обнаружен конец файла. Если во входном потоке встречается символ *delim*, он *не* считывается, а остается в потоке до следующей операции чтения. Функция автоматически добавляет в конец массива нулевой символ и возвращает ссылку на поток.

Функция **get(streambuf &buf)** считывает символы из входного потока в объект класса **streambuf**. Символы считываются, пока не встретится символ перехода на новую строку или будет обнаружен конец файла. Если во входном потоке обнаружен символ перехода на новую строку, он *не* считывается. Функция возвращает ссылку на поток.

Функция **get(streambuf &buf, char delim)** считывает символы из входного потока в объект класса **streambuf**. Символы считываются, пока не встретится разделитель ***delim***

или будет обнаружен конец файла. Если во входном потоке обнаружен символ ***delim***, он считывается, но не записывается в массив *buf*. Функция возвращает ссылку на поток.

Зависимые функции: **get()** и **read()**.

Функция good

```
#include <iostream>
bool good() const;
```

Функция **good()** является членом класса **ios**.

Если при выполнении операции ввода или вывода в потоке никаких ошибок не обнаружено, функция **good()** возвращает значение **true**, в противном случае возвращается значение **false**.

Зависимые функции: **bad()**, **fail()**, **eof()**, **clear()** и **rdstate()**.

Функция ignore

```
#include <iostream>
istream &ignore(streamsize num = 1, int delim = EOF);
```

Функция **ignore()** является членом класса **istream**.

Эта функция считывает и отбрасывает символы из входного потока, пока не будут считаны и проигнорированы *num*−1 символов (по умолчанию — один), либо пока не встретится символ *delim* (по умолчанию — **EOF**). Обнаруженный разделитель удаляется из входного потока. Функция возвращает ссылку на поток.

Зависимые функции: **get()** и **getline()**.

Функция open

```
#include <fstream>
void fstream::open(const char *filename,
                   ios::openmode mode = ios::in | ios::out);
void ifstream::open(const char *filename,
                   ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
                   ios::openmode mode = ios::out | ios::trunc);
```

Функция **open()** является членом классов **fstream**, **ifstream** и **ofstream**. Она связывает файл с потоком. Параметр *filename* задает имя класса, которое может включать в себя путь к файлу. Значение *mode* определяет режим, в котором открывается файл. Этот параметр может принимать одно или несколько значений, перечисленных ниже.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Эти значения можно комбинировать с помощью логической операции OR.

Параметр **ios::app** означает, что все данные будут записываться в конец существующего файла. Это возможно, лишь если файл открыт для вывода. Значение **ios::ate** устанавливает файловый курсор на конец открываемого файла. При этом операции ввода-вывода могут выполняться в любой позиции файла.

Параметр **ios::binary** означает, что файл открывается в бинарном режиме. По умолчанию файл открывается в текстовом режиме.

Параметр **ios::in** означает, что файл открыт для ввода. Значение **ios::out** свидетельствует, что файл открыт для вывода. В то же время открытие потока **ifstream** приводит к автоматическому открытию файла для ввода, открытие потока **ofstream** приводит к автоматическому открытию файла для вывода, а открытие потока **fstream** приводит к автоматическому открытию файла для ввода и вывода.

Параметр **ios::trunc** означает, что содержимое существующего файла, имя которого совпадает со значением параметра *filename*, будет уничтожено, а длина файла будет усечена (truncated) до нуля.

Во всех случаях отказа функции **open()** возвращается значение **false**. Таким образом, перед использованием файла, следует убедиться, что он успешно открыт.

Зависимые функции: **close()**, **fstream()**, **ifstream()** и **ofstream()**.

Функция peek

```
#include <iostream>
int peek();
```

Функция **peek()** является членом класса **istream**.

Эта функция возвращает следующий символ из входного потока или признак конца файла **EOF**. Функция ни при каких обстоятельствах не удаляет символ из потока.

Зависимая функция: **get()**.

Функция precision

```
#include <iostream>
streamsize precision() const;
streamsize precision(streamsize p);
```

Функция **precision()** является членом класса **ios**, унаследованным от класса **ios_base**.

По умолчанию значения с плавающей точкой выводятся с шестью цифрами после десятичной точки. Однако, используя второй вариант функции **precision()**, можно вывести на экран *p* цифр после десятичной точки. Функция возвращает исходное значение объекта класса **streamsize**.

Первый вариант функции возвращает текущее значение объекта класса **streamsize**.

Зависимые функции: **width()** и **fill()**.

Функция put

```
#include <iostream>
ostream &put(char ch);
```

Функция **put()** является членом класса **ostream**. Она записывает символ *ch* в соответствующий поток вывода и возвращает ссылку на поток.

Зависимые функции: **write()** и **get()**.

Функция putback

```
#include <iostream>
istream &putback(char ch);
```

Функция **putback()** является членом класса **istream**. Она возвращает символ *ch* в соответствующий поток ввода.

Зависимая функция: **peek()**.

Функция rdstate

```
#include <iostream>
iostate rdstate() const;
```

Функция **rdstate()** является членом класса **ios**. Она возвращает состояние соответствующего потока. Система ввода-вывода сохраняет информацию о результате каждой операции, связанной с активным потоком. Текущее состояние потока содержится в объекте типа **iostate**, в котором определены следующие флаги.

Имя	*Значение*
goodbit	Никаких ошибок не обнаружено.
eofbit	Обнаружен конец файла.
failbit	Обнаружена поправимая ошибка ввода-вывода.
badbit	Обнаружена непоправимая ошибка ввода-вывода.

Эти флаги заданы в классе **ios** в качестве элементов перечисления, унаследованных от класса **ios_base**.

Если никаких ошибок не обнаружено, функция **rdstate()** возвращает флаг **goodbit**, в противном случае устанавливается бит, соответствующий возникшей ошибке.

Зависимые функции: **eof()**, **good()**, **clear()**, **setstate()** и **fail()**.

Функция read

```
#include <iostream>
istream &read(char *buf, streamsize num);
```

Функция **read()** является членом класса **istream**. Она считывает из входного потока *num* байтов и записывает их в буфер *buf*. Если конец файла обнаруживается раньше времени, функция **read()** прекращает работу, устанавливает флаги **failbit**, а считанные символы остаются в буфере (см. раздел, посвященный функции **gcount()**). Функция **read()** возвращает ссылку на поток.

Зависимые функции: **gcount()**, **readsome()**, **get()**, **getline()** и **write()**.

Функция readsome

```
#include <iostream>
streamsize readsome(char *buf, streamsize num);
```

Функция **readsome()** является членом класса **istream**. Она считывает из входного потока *num* байтов и записывает их в буфер *buf*. Если поток содержит меньше *num* символов, они все считываются. Функция **readsome()** возвращает количество считанных символов. Разница между функциями **read()** и **readsome()** заключается в том, что функция **readsome()** не устанавливает флаги **failbit**, если поток содержит меньше *num* символов.

Зависимые функции: **gcount()**, **read()** и **write()**.

Функции seekg и seekp

```
#include <iostream>
istream &seekg(off_type offset, ios::seekdir origin);
istream &seekg(pos_type position);

ostream &seekp(off_type offset, ios::seekdir origin);
ostream &seekp(pos_type position);
```

Функция **seekg()** является членом класса **istream**, а функция **seekp()** принадлежит классу **ostream**.

Эти функции обеспечивают произвольный доступ к файлу. Для этого система ввода-вывода содержит два указателя, связанных с файлом. Один из них является *указателем ввода* (get pointer). Он ссылается на позицию, с которой начинается следующая операция ввода. Другой указатель ссылается на позицию, с которой начинается следующая операция вывода. Он называется *указателем вывода* (put pointer). При выполнении очередной операции ввода-вывода соответствующий указатель последовательно продвигается вперед. Однако с помощью функций **seekg()** и **seekp()** можно перемещаться по файлу не последовательно, а произвольно.

Вариант функции **seekg()** с двумя параметрами перемещает указатель ввода на *offset* байтов относительно позиции *origin*. Параметр *offset* имеет тип **off_type**, позволяющий хранить наибольшее возможное значение, которое может принимать параметр *offset*.

Параметр *origin* имеет тип **seekdir** и представляет собой перечисление, содержащее следующие значения.

ios::beg	Перемещение от начала.
ios::cur	Перемещение от текущей позиции.
ios::end	Перемещение от конца.

Версии функций **seekg()** и **seek(p)**, имеющие по одному параметру, перемещают файловый указатель на позицию, заданную параметром *position*. Значение этого параметра должно быть предварительно получено с помощью функций **tellg()** или **tellp()** соответственно. Тип **pos_type** позволяет хранить наибольшее возможное значение, которое может принимать параметр *position*. Эти функции возвращают ссылку на соответствующий поток.

Зависимые функции: **tellg()** и **tellp()**.

Функция setf

```
#include <iostream>
fmtflags setf(fmtflags flags);
fmtflags setf(fmtflags flags1, fmtflags flags2);
```

Функция **setf()** является членом класса **ios**, унаследованным от класса **ios_base**. Она устанавливает флаги форматирования, связанные с потоком. Значения флагов форматирования описаны в предыдущих разделах.

Первая версия функции **setf()** возвращает флаги форматирования, заданные параметром *flags*. (Все другие флаги игнорируются.) Например, для того, чтобы установить флаг **showpos** для потока **cout**, можно выполнить следующий оператор.

```
cout.setf(ios::showpos);
```

Если необходимо установить несколько флагов, используется логическая операция OR.

Вызов функции **setf()** всегда связан с конкретным потоком. Эта функция никогда не вызывается сама по себе. Иначе говоря, в языке C++ не существует глобального форматирования. Каждый поток хранит информацию о форматировании отдельно.

Вторая версия функции **setf()** влияет только на флаги, установленные в параметре *flags2*. Соответствующие флаги сначала сбрасываются, а затем устанавливаются вновь в соответствии с флагами, заданными в параметре *flags1*. Если параметр *flags1* содержит другой набор флагов, устанавливаются лишь те флаги, которые заданы в параметре *flags2*.

Обе версии функции **setf()** возвращают предыдущее состояние флагов форматирования, связанных с потоком.

Зависимые функции: **unsetf()** и **flags()**.

Функция setstate

```
#include <iostream>
void setstate(iostate flags) const;
```

Функция **setstate()** является членом класса **ios**. Она устанавливает состояние соответствующего потока, описанное параметром *flags* (см. раздел, посвященный функции **rdstate()**).

Зависимые функции: **clear()** и **rdstate()**.

Функция str

```
#include <iostream>
string str() const;
void str(string &s);
```

Функция **str()** является членом классов **stringstream**, **istringstream** и **ostringstream**.

Первая версия функции **str()** возвращает объект класса **string**, в котором хранится текущее содержание строкового потока.

Вторая версия удаляет из строкового потока текущую строку и заменяет ее строкой *s*.

Зависимые функции: **get()** и **put()**.

Функции stringstream, istringstream и ostringstream

```
#include <sstream>
explicit stringstream::open(ios::openmode mode = ios::in | ios::out);
explicit stringstream::open(const string &str,
                  ios::openmode mode = ios::in | ios::out);
explicit istringstream::open(ios::openmode mode = ios::in);
explicit istringstream::open(const string str,
                  ios::openmode mode = ios::in);
explicit ostringstream::open(ios::openmode mode = ios::out);
explicit ostringstream::open(const string str,
                  ios::openmode mode = ios::out);
```

Функции **stringstream()**, **istringstream()** и **ostringstream()** являются конструкторами классов **stringstream**, **istringstream** и **ostringstream** соответственно.

Версии **stringstream()**, **istringstream()** и **ostringstream()**, имеющие единственный параметр **openmode**, создают пустые потоки. Версии, имеющие параметр класса **string**, инициализируют строковые потоки.

Рассмотрим пример, демонстрирующий применение строкового потока.

```
// Демонстрация строковых потоков.
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
  stringstream s("Это — исходная строка.");

  // Вводим строку.
  string str =- s.str();
  cout << str << endl;
```

```
    // Выводим данные в строковый поток.
    s << "Числа: " << 10 << " " << 123.2;

    int i;
    double d;
    s >. str >> i >> d;
    cout << str << " " << i << " " << d;

    return 0;
}
```

Результаты работы этой программы приведены ниже

```
Это — исходная строка.
Числа: 10 123.2
```

Зависимая функция: **str()**.

Функция sync_with_stdio

```
#include <iostream>
bool sync_with_stdio(bool sync = true);
```

Функция **sync_with_stdio()** является членом класса **ios**, унаследованным от класса **ios_base**.

Вызов функции **sync_with_stdio()** позволяет безопасно использовать систему ввода-вывода языка С наряду с объектно-ориентированной системой ввода-вывода, основанной на классах. Чтобы отключить синхронизацию с потоком **stdio**, следует передать функции **sync_with_stdio()** значение **false**. Функция возвращает предыдущее значение: **true** — если синхронизация установлена, и **false** — если нет. По умолчанию стандартные потоки синхронизируются. Функция **sync_with_stdio()** работает надежно, только если она вызвана до выполнения каких-либо операций ввода-вывода.

Функции tellg и tellp

```
#include <iostream>
pos_type tellg();
pos_type tellp();
```

Функция **tellg()** является членом класса **istream**, а функция **tellp()** принадлежит классу **ostream**.

Система ввода-вывода языка C++ управляет двумя указателями, ссылающимися на позиции внутри файла. Один из них является *указателем ввода*. Он ссылается на позицию, с которой начинается следующая операция ввода. Другой указатель ссылается на позицию, с которой начинается следующая операция вывода. Он называется *указателем вывода*. При выполнении очередной операции ввода-вывода соответствующий указатель последовательно продвигается вперед. С помощью функций **tellg()** и **tellp()** можно определить текущее положение указателей ввода и вывода соответственно.

Тип **pos_type** позволяет хранить наибольшее возможное значение, которое может возвращаться функциями **tellg()** и **tellp()**. Значения, возвращаемые этими функциями, используются соответственно функциями **seekg()** и **seekp()**.

Зависимые функции: **seekg()** и **seekp()**.

Функция unsetf

```
#include <iostream>
void unsetf(fmtflags flags);
```

Функция **unsetf()** является членом класса **ios**, унаследованным от класса **ios_base**. Она сбрасывает один или несколько флагов форматирования, заданных параметром *flags*. (Все другие флаги игнорируются.)

Зависимые функции: **setf()** и **flags()**.

Функция width

```
#include <iostream>
streamsize width() const;
streamsize width(streamsize w);
```

Функция **width()** является членом класса **ios**, унаследованным от класса **ios_base**.

Первая версия функции используется, чтобы определить текущую ширину поля вывода. Для того чтобы изменить ширину поля, следует вызвать второй вариант функции **width()**.

Зависимые функции: **precision()** и **fill()**.

Функция write

```
#include <iostream>
ostream &write(const char *buf, streamsize num);
```

Функция **write()** является членом класса **ostream**. Она записывает *num* байтов из входного потока в буфер *buf*. Функция возвращает ссылку на поток.

Зависимые функции: **read()** и **put()**.

Полный справочник по

Глава 33

Стандартные контейнерные классы

В этой главе описываются контейнерные классы, определенные в библиотеке STL. Контейнеры предназначены для хранения других объектов. Они выделяют память, необходимую для хранения объектов, и определяют механизм доступа к ним. Таким образом, контейнеры представляют собой довольно сложное понятие.

На заметку *Обзор библиотеки STL содержится в главе 24.*

При описании контейнеров мы будем соблюдать следующие соглашения. Для ссылок на различные итераторы используются названия, перечисленные ниже.

Упоминая типы итераторов, использованные в описаниях шаблонов, мы будем придерживаться такой терминологии.

Термин	*Смысл*
`BiIter`	Двунаправленный итератор
`ForIter`	Прямой итератор
`InIter`	Итератор ввода
`OutIter`	Итератор вывода
`RandIter`	Итератор произвольного доступа

Унарные предикаты относятся к типу `UnPred`, а бинарные — к типу `BinPred`. Аргументы бинарных предикатов всегда перечисляются по порядку: *first*, *second*. В качестве аргументов как унарных, так и бинарных предикатов, должны использоваться лишь объекты, хранящиеся в контейнере.

Функции сравнения обозначаются термином `Comp`.

И еще одно замечание: если в описании сказано, что итератор указывает на конец контейнера, это значит, что итератор установлен на объект, непосредственно следующий за последним объектом контейнера.

Контейнерные классы

В библиотеке STL определены следующие контейнеры.

Контейнер	*Описание*	*Заголовок*
`bitset`	Набор битов.	`<bitset>`
`deque`	Двусторонняя очередь.	`<deque>`
`list`	Линейный список.	`<list>`
`map`	Хранит пары ключ-значение, в которых каждому ключу соответствует лишь одно значение.	`<map>`
`multimap`	Хранит пары ключ-значение, в которых каждому ключу может соответствовать несколько значений.	`<map>`
`multiset`	Множество, в котором каждый элемент может входить несколько раз.	`<set>`
`priority_queue`	Очередь с приоритетами.	`<queue>`
`queue`	Очередь.	`<queue>`
`set`	Множество, в котором каждый элемент входит лишь один раз.	`<set>`
`stack`	Стек.	`<stack>`
`vector`	Динамический массив.	`<vector>`

Каждый из контейнеров будет описан в следующих разделах. Поскольку все контейнеры реализованы с помощью шаблонных классов, они могут хранить данные любых типов. В описаниях эти типы представлены обобщенным типом `T`.

Поскольку имена типов данных, хранящихся в контейнере, произвольны, контейнерные классы содержат оператор **typedef**, конкретизирующий имена типов. Рассмотрим имена типов, используемых в контейнерных классах.

size_type	Некая разновидность целочисленного типа
reference	Ссылка на элемент.
const_reference	Константная ссылка на элемент.
difference_type	Разность между двумя адресами
iterator	Итератор.
const_iterator	Константный итератор.
reverse_iterator	Обратный итератор.
const_reverse_iterator	Константный обратный итератор.
value_type	Тип значения, хранящегося в контейнере.
allocator_type	Тип распределителя.
key_type	Тип ключа.
key_compare	Тип функции, сравнивающей два ключа.
value_compare	Тип функции, сравнивающей два значения.
pointer	Указатель
const_pointer	Константный указатель
container_type	Контейнер

Класс bitset

Класс **bitset** выполняет операции над наборами битов. Его шаблонная спецификация имеет следующий вид.

```
template <size_t N> class bitset;
```

Здесь параметр *N* задает количество битов в наборе. Класс **bitset** имеет следующие конструкторы.

```
bitset();

bitset(unsigned long bits);

explicit bitset(const string &s, size_t i = 0, size_t num = npos);
```

Первый вариант конструктора создает пустой набор битов. Второй вариант создает набор битов, установленных в соответствии с параметром *bits*. Третий конструктор создает набор битов, используя строку *s*, начиная с позиции *i*. Эта строка должна содержать только нули и единицы. При установке битов используются либо *num*, либо *s*.**size()**−*i* битов, в зависимости от того, какое из этих чисел меньше. Константа **npos** представляет собой значение, достаточно большое для того, чтобы задать максимальную длину строки *s*.

В классе **bitset** определены операторы "<<" и ">>".

Класс **bitset** содержит следующие функции-члены.

Функция-член	*Описание*
bool any() const;	Возвращает значение **true**, если вызывающий набор битов состоит только из единиц, в противном случае возвращает значение **false**.
size_t count() const;	Возвращает количество единичных битов.
bitset<N> &flip();	Инвертирует все биты в вызывающем наборе и возвращает указатель ***this**.
bitset<N> &flip(size_t *i*);	Инвертирует бит в *i*-й позиции вызывающего набора и возвращает указатель ***this**.

Функция-член	*Описание*
`bool none() const;`	Возвращает значение `true`, если все биты в наборе являются нулевыми.
`bool operator!= (const bitset<N> &op2) const;`	Возвращает значение `true`, если вызывающий набор битов отличается от набора *op2*, указанного в правой части оператора.
`bool operator== (const bitset<N> &op2) const;`	Возвращает значение `true`, если вызывающий набор битов совпадает с набором, заданным в правой части оператора *op2*.
`bitset<N> &operator&= (const bitset<N> &op2);`	Применяет операцию `AND` к каждой паре битов, принадлежащих вызывающему набору и набору, заданному в правой части оператора *op2*, соответственно. Возвращает указатель `*this`, оставляя результат в вызывающем наборе битов.
`bitset<N> &operator^!= (const bitset<N> &op2);`	Применяет операцию `XOR` к каждой паре битов, принадлежащих вызывающему набору и набору, заданному в правой части оператора *op2*, соответственно. Возвращает указатель `*this`, оставляя результат в вызывающем наборе битов.
`bitset<N> &operator|= (const bitset<N> &op2);`	Применяет операцию `OR` к каждой паре битов, принадлежащих вызывающему набору и набору, заданному в правой части оператора *op2*, соответственно. Возвращает указатель `*this`, оставляя результат в вызывающем наборе битов.
`bitset<N> &operator~() const;`	Инвертирует все биты в вызывающем наборе битов и возвращает результат.
`bitset<N> &operator<<=(size_t num);`	Сдвигает все биты вызывающего набора на *num* позиций влево. Возвращает указатель `*this`, оставляя результат в вызывающем наборе битов.
`bitset<N> &operator>>=(size_t num);`	Сдвигает все биты вызывающего набора на *num* позиций вправо. Возвращает указатель `*this`, оставляя результат в вызывающем наборе битов.
`reference operator[](size_t i);`	Возвращает ссылку на *i*-й бит в вызывающем наборе.
`bitset<N> &reset();`	Сбрасывает все биты вызывающего набора и возвращает указатель `*this`.
`bitset<N> &reset(size_t i);`	Сбрасывает *i*-й бит вызывающего набора и возвращает указатель `*this`.
`bitset<N> &set();`	Устанавливает все биты вызывающего набора и возвращает указатель `*this`.
`bitset<N> &set (size_t i, int val = 1);`	Присваивает *i*-му биту вызывающего набора значение *val* и возвращает указатель `*this`. Любое ненулевое значение параметра *val* трактуется как единица.
`size_t size() const;`	Возвращает количество битов, которые могут храниться в битовом наборе.
`bool test(size_t i) const;`	Возвращает состояние *i*-го бита.
`string to_string() const;`	Возвращает строку, содержащую битовую маску вызывающего битового набора.
`unsigned long to_ulong() const;`	Преобразует вызывающий битовый набор в длинное целое число без знака.

Класс deque

Класс **deque** обеспечивает работу с двусторонней очередью. Его шаблонная спецификация выглядит следующим образом.

```
template <class T, class Allocator = allocator<T> > class deque
```

Здесь **T** обозначает тип данных, хранящихся в классе **deque**. Этот класс содержит следующие конструкторы.

```
explicit deque(const Allocator &a = Allocator());

explicit deque(size_type num, const T &val = T(),
   const Allocator &a = Allocator());

deque(const deque<T, Allocator> &ob);

template <class InIter> deque(InIter start, InIter end,
   const Allocator &a = Allocator());
```

Первый конструктор создает пустую двустороннюю очередь. Второй конструктор создает двустороннюю очередь, содержащую *num* элементов, имеющих значение *val*. Третий конструктор создает двустороннюю очередь, содержащую элементы объекта *ob*. Четвертый конструктор создает двустороннюю очередь, содержащую элементы, изменяющиеся в диапазоне от *start* до *end*.

Кроме того, в классе **deque** определены операции

```
==, <, <=, !=, >, >=
```

Класс **deque** содержит следующие функции-члены.

Функция-член	*Описание*
`template <class InIter> void assign(InIter start, InIter end);`	Присваивает двусторонней очереди последовательность элементов, определенную итераторами *start* и *end*.
`void assign(size_type num, const T &val);`	Присваивает двусторонней очереди *num* элементов, имеющих значение *val*.
`reference at(size_type i);` `const_reference at(size_type i) const;`	Возвращает ссылку на *i*-й элемент двусторонней очереди.
`reference back();` `const_reference back() const;`	Возвращает итератор, установленный на последний элемент двусторонней очереди.
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент двусторонней очереди.
`void clear();`	Удаляет все элементы из двусторонней очереди.
`bool empty() const;`	Возвращает значение **true**, если вызывающая двусторонняя очередь пуста, и значение **false** — в противном случае.
`const_iterator end() const;` `iterator end();`	Возвращает итератор, установленный на конец двусторонней очереди.
`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`iterator erase(iterator start, iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.

Окончание табл.

Функция-член	Описание
`reference front();` `const_reference front() const;`	Возвращает итератор, установленный на первый элемент двусторонней очереди.
`allocator_type get_allocator() const;`	Возвращает распределитель памяти для двусторонней очереди.
`iterator insert(iterator i, const T &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.
`void insert(iterator i, size_type num,` `const T &val);`	Вставляет *num* копий значения *val* непосредственно перед элементом, на который ссылается итератор *i*.

`template <class InIter>`
`void insert(iterator i,`
`InIter start, InIter end);`
	Вставляет элементы последовательности, определенной итераторами *start* и *end*, непосредственно перед элементом, на который ссылается итератор *i*.

`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в двусторонней очереди.

`reference operator[](size_type i);`
`const_reference operator[]`
`(size_type i) const;`
	Возвращает ссылку на *i*-й элемент.

`void pop_back();`	Удаляет последний элемент двусторонней очереди.
`void pop_front();`	Удаляет первый элемент двусторонней очереди.
`void push_back(const T &val);`	Добавляет элемент, имеющий значение *val*, в конец двусторонней очереди.
`void push_front(const T &val);`	Добавляет элемент, имеющий значение *val*, в начало двусторонней очереди.
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный на конец двусторонней очереди.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начало двусторонней очереди.
`void resize(size_type num, T val=T());`	Меняет размер двусторонней очереди на значение, заданное параметром *num*. Если очередь необходимо удлинить, элементы, имеющие значение *val*, добавляются в конец.
`size_type size() const;`	Возвращает текущее количество элементов, находящихся в очереди.
`void swap(deque<T, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающей очереди, и объект *ob*.

Класс list

Класс **list** обеспечивает работу со списком. Его шаблонная спецификация выглядит следующим образом.

```
template <class T, class Allocator = allocator<T> > class list
```

Здесь **T** обозначает тип данных, хранящихся в классе **list**. Этот класс содержит следующие конструкторы.

```
explicit list(const Allocator &a = Allocator());

explicit list(size_type num, const T &val = T(),
   const Allocator &a = Allocator());

list(const list<T, Allocator> &ob);
```

```
template <class InIter> list(InIter start, InIter end,
   const Allocator &a = Allocator());
```

Первый конструктор создает пустой список. Второй конструктор создает список, содержащий *num* элементов, имеющих значение *val*. Третий конструктор создает список, содержащий элементы объекта *ob*. Четвертый конструктор создает список, содержащий элементы, изменяющиеся в диапазоне от *start* до *end*.

Кроме того, в классе **list** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **list** содержит такие функции-члены.

Функция-член	*Описание*
`template <class InIter> void assign(InIter start, InIter end);`	Присваивает списку последовательность элементов, определенную итераторами *start* и *end*.
`void assign(size_type num, const T &val);`	Присваивает списку *num* элементов, имеющих значение *val*.
`reference back();` `const_reference back() const;`	Возвращает итератор, установленный на последний элемент списка.
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент списка.
`void clear();`	Удаляет все элементы из списка.
`bool empty() const;`	Возвращает значение **true**, если вызывающий список пуст, и значение **false** — в противном случае.
`const_iterator end() const;` `iterator end();`	Возвращает итератор, установленный на конец списка.
`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`iterator erase(iterator start,` `iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`reference front();` `const_reference front() const;`	Возвращает итератор, установленный на первый элемент списка.
`allocator_type get_allocator() const;`	Возвращает распределитель памяти для списка.
`iterator insert` `(iterator i, const T &val=T());`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.
`void insert(iterator i, size_type num,` `const T &val);`	Вставляет *num* копий значения *val* непосредственно перед элементом, на который ссылается итератор *i*.

`template <class InIter>`
`void insert(iterator i,`
`InIter start, InIter end);`
	Вставляет элементы последовательности, определенной итераторами *start* и *end*, непосредственно перед элементом, на который ссылается итератор *i*.

`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в списке.

`void merge(list<T,Allocator> &ob);`
`template <class Comp>`
`void merge(<list<T,Allocator>`
`&ob, Comp cmpfn);`
	Объединяет упорядоченный список, содержащийся в объекте *ob*, и вызывающий упорядоченный список. В результате возникает упорядоченный список. После объединения список, содержащийся в объекте *ob*, пуст. Вторая форма функции использует заданную функцию сравнения двух элементов.

Окончание табл.

Функция-член	Описание
`void pop_back();`	Удаляет последний элемент списка.
`void pop_front();`	Удаляет первый элемент списка.
`void push_back(const T &val);`	Добавляет элемент, имеющий значение *val*, в конец списка.
`void push_front(const T &val);`	Добавляет элемент, имеющий значение *val*, в начало списка.
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный в конце списка.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начале списка.
`void remove(const T &val);`	Удаляет из списка все элементы, имеющие значение *val*.
`template <class UnPred>` `void remove_if(UnPred pr);`	Удаляет из списка все элементы, если унарный предикат *pr* имеет истинное значение.
`void resize(size_type num, T val=T());`	Меняет размер списка на значение, заданное параметром *num*. Если список необходимо удлинить, элементы, имеющие значение *val*, добавляются в конец.
`void reverse();`	Меняет порядок следования элементов списка на противоположный.
`size_type size() const;`	Возвращает текущее количество элементов, находящихся в списке.

`void sort();`
`template <class Comp>`
`void sort(Comp cmpfn);`
	Упорядочивает список. Второй вариант функции упорядочивает список, используя заданную функцию сравнения *cmpfn*.

`void splice(iterator i,` `list<T, Allocator> &ob);`	Вставляет содержимое объекта *ob* в позицию списка, заданную итератором *i*. После выполнения операции объект *ob* становится пустым.

`void splice(iterator i,`
`list<T, Allocator> &ob,`
`iterator el);`
	Удаляет из объекта *ob* элемент, на который установлен итератор *el*, и вставляет его в позицию списка, заданную итератором *i*.

`void splice(iterator i,`
`list<T, Allocator> &ob,`
`iterator start, iterator end);`
	Удаляет из объекта *ob* элементы, лежащие в диапазоне, определенном итераторами *start* и *end*, и вставляет их в позицию списка, заданную итератором *i*.

`void swap(list<T, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем списке, и элементы из объекта *ob*.

`void unique();`
`template <class BinPred>`
`void unique(BinPred pr);`
	Удаляет дубликаты из вызывающего списка. Второй вариант функции использует для определения уникальности элемента бинарный предикат *pr*.

Класс map

Класс **map** обеспечивает работу с ассоциативным контейнером, в котором каждому значению соответствует уникальный ключ. Его шаблонная спецификация выглядит следующим образом.

```
template <class Key, class T, class Comp = less<Key>,
   class Allocator = allocator<pair<const Key, T> > > class map
```

Здесь **Key** означает тип ключа, а **T** — тип данных, хранящихся в классе **map**, **Comp** представляет собой функцию, предназначенную для сравнения двух ключей. Этот класс содержит следующие конструкторы.

```
explicit map(const Comp &cmpfn = Comp(),
             const Allocator &a = Allocator());

map(const map<Key, T, Comp, Allocator> &ob);

template <class InIter> map(InIter start, InIter end,
   const Comp &cmpfn = Comp(),
   const Allocator &a = Allocator());
```

Первый конструктор создает пустой ассоциативный массив. Второй конструктор создает ассоциативный массив, содержащий элементы объекта *ob*. Третий конструктор создает ассоциативный массив, содержащий элементы, изменяющиеся в диапазоне от *start* до *end*. Параметр *cmpfn* задает функцию, определяющую порядок следования элементов в ассоциативном массиве.

Кроме того, в классе **map** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **map** содержит такие функции-члены В этом описании **key_type** представляет собой тип ключа, а **value_type** обозначает пару **pair<Key, T>**.

Функция-член	*Описание*
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент ассоциативного массива.
`void clear();`	Удаляет все элементы из ассоциативного массива
`size_type count(const key_type &k) const;`	Возвращает количество экземпляров ключа *k* в ассоциативном массиве (1 или 0).
`bool empty() const;`	Возвращает значение **true**, если вызывающий ассоциативный массив пуст, и значение **false** — в противном случае.
`iterator end();` `const_iterator end() const;`	Возвращает итератор, установленный на конец ассоциативного массива.

`pair<iterator,iterator>`
`  equal_range(const key_type &k);`
`pair<const_iterator,const_iterator>`
`  equal_range(const key_type &k) const;`
	Возвращает пару итераторов, установленных на первый и последний элементы ассоциативного массива, содержащих указанный ключ.

`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`iterator erase(iterator start,` `  iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`size_type erase(const key_type &k);`	Удаляет из ассоциативного массива элементы, имеющие ключ со значением *k*.

`iterator find(const key_type &k);`
`const_iterator find(const key_type &k)`
`  const;`
	Возвращает итератор, установленный на заданный ключ. Если ключ не найден, возвращается итератор, ссылающийся на конец массива.

`allocator_type get_allocator() const;`	Возвращает распределитель памяти для ассоциативного массива.

Окончание табл.

Функция-член	Описание
`iterator insert(iterator i, const T &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.
`template <class InIter> void insert(InIter start, InIter end);`	Вставляет элементы последовательности, определенной итераторами *start* и *end*.
`pair<iterator,bool> insert(const value_type &val);`	Вставляет значение *val* в вызывающий ассоциативный массив. Возвращает итератор, установленный на вставленный элемент. Элемент вставляется, только если его в массиве еще не было. Если элемент вставлен успешно, возвращается пара `pair<iterator,true>`, в противном случае возвращается пара `pair<iterator,false>`.
`key_compare key_comp() const;`	Возвращает функтор, сравнивающий ключи.
`iterator lower_bound(const key_type &k,` `const_iter lower_bound (const key_type &k) const;`	Возвращает итератор, установленный на первый элемент ассоциативного массива, ключ которого равен значению *k* или превышает его.
`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в ассоциативном массиве.
`mapped_type &operator[] (const key_type &i);`	Возвращает ссылку на элемент, заданный параметром *i*. Если такого элемента не существует, он вставляется в ассоциативный массив.
	Удаляет последний элемент списка.
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный в конце ассоциативного массива.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начале ассоциативного массива
`size_type size() const;`	Возвращает текущее количество элементов ассоциативного массива.
`void swap(map<Key,T,Comp, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем ассоциативном массиве, и элементы из объекта *ob*.
`iterator upper_bound(const key_type &i);` `const_iterator upper_bound (const key_type &i) const;`	Возвращает итератор. установленный на первый элемент ассоциативного массива, ключ которого превышает значение *k*.
`value_compare value_comp() const;`	Возвращает функтор, сравнивающий два значения.

Класс multimap

Класс **multimap** обеспечивает работу с ассоциативным контейнером, в котором каждому значению могут соответствовать несколько ключей. Его шаблонная спецификация выглядит следующим образом.

```
template <class Key, class T, class Comp = less<Key>,
   class Allocator = allocator<pair<const Key, T> > > class multimap
```

Здесь **Key** означает тип ключа, а **T** — тип данных, хранящихся в классе **multimap**, **Comp** представляет собой функцию, предназначенную для сравнения двух ключей. Этот класс содержит следующие конструкторы.

```
explicit multimap(const Comp &cmpfn = Comp(),
                  const Allocator &a = Allocator());

multimap(const multimap<Key, T, Comp, Allocator> &ob);

template <class InIter> multimap(InIter start, InIter end,
   const Comp &cmpfn = Comp(),
   const Allocator &a = Allocator());
```

Первый конструктор создает пустой ассоциативный массив, а второй — ассоциативный массив, содержащий элементы объекта *ob*. Третий конструктор создает ассоциативный массив, содержащий элементы, изменяющиеся в диапазоне от *start* до *end*. Параметр *cmpfn* задает функцию, определяющую порядок следования элементов в ассоциативном массиве.

Кроме того, в классе **multimap** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **multimap** содержит такие функции-члены. В этом описании **key_type** представляет собой тип ключа, а **value_type** обозначает пару **pair<Key, T>**.

Функция-член	*Описание*
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент ассоциативного массива.
`void clear();`	Удаляет все элементы из ассоциативного массива.
`size_type count(const key_type &k) const;`	Возвращает количество экземпляров ключа *k* в ассоциативном массиве.
`bool empty() const;`	Возвращает значение **true**, если вызывающий ассоциативный массив пуст, и значение **false** — в противном случае.
`const_iterator end() const;` `iterator end();`	Возвращает итератор, установленный на конец ассоциативного массива.

`pair<iterator,iterator>`
`  equal_range(const key_type &k);`
`pair<const_iterator,const_iterator>`
`  equal_range(const key_type &k) const;`
	Возвращает пару итераторов, установленных на первый и последний элементы ассоциативного массива, содержащих указанный ключ.

`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`iterator erase(iterator start, iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`size_type erase(const key_type &k);`	Удаляет из ассоциативного массива элементы, имеющие ключ со значением *k*.
`iterator find(const key_type &k);` `const_iterator find(const key_type &k) const;`	Возвращает итератор, установленный на заданный ключ. Если ключ не найден, возвращается итератор, ссылающийся на конец массива.
`allocator_type get_allocator() const;`	Возвращает распределитель памяти для ассоциативного массива.

Функция-член	Описание
`iterator insert(iterator i, const value_type &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.
`template <class InIter> void insert(InIter start, InIter end);`	Вставляет элементы последовательности, определенной итераторами *start* и *end*.
`pair<iterator,bool> insert(const value_type &val);`	Вставляет значение *val* в вызывающий ассоциативный массив.
`key_compare key_comp() const;`	Возвращает функтор, сравнивающий ключи.
`iterator lower_bound(const key_type &k), const_iterator lower_bound (const key_type &k) const;`	Возвращает итератор, установленный на первый элемент ассоциативного массива, ключ которого равен или превышает значение *k*.
`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в ассоциативном массиве.
`reverse_iterator rbegin(); const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный в конце ассоциативного массива.
`reverse_iterator rend(); const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начале ассоциативного массива.
`size_type size() const;`	Возвращает текущее количество элементов ассоциативного массива.
`void swap(map<Key,T,Comp, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем ассоциативном массиве, и элементы из объекта *ob*.
`iterator upper_bound(const key_type &i); const_iterator upper_bound(const key_type &i) const;`	Возвращает итератор, установленный на первый элемент ассоциативного массива, ключ которого превышает значение *k*.
`value_compare value_comp() const;`	Возвращает функтор, сравнивающий два значения.

Класс multiset

Класс **multiset** обеспечивает работу с мультимножеством, в котором каждому значению могут соответствовать несколько ключей. Его шаблонная спецификация выглядит следующим образом.

```
template <class Key, class Comp = less<Key>,
   class Allocator = allocator< Key, > > class multiset
```

Здесь **Key** означает тип ключа, а **allocator< Key Comp** представляет собой функцию, предназначенную для сравнения двух ключей. Этот класс содержит следующие конструкторы.

```
explicit multiset(const Comp &cmpfn = Comp(),
             const Allocator &a = Allocator());

multimap(const multiset<Key, Comp, Allocator> &ob);

template <class InIter> multiset(InIter start, InIter end,
   const Comp &cmpfn = Comp(),
   const Allocator &a = Allocator());
```

Первый конструктор создает пустое мультимножество. Второй конструктор создает мультимножество, содержащее элементы объекта *ob*. Третий конструктор создает мультимножество, содержащее элементы, расположенные в диапазоне от *start* до *end*. Параметр *cmpfn* задает функцию, определяющую порядок следования элементов в мультимножестве.

Кроме того, в классе **multiset** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **multiset** содержит такие функции-члены. В этом описании **key_type** представляет собой тип ключа, а **value_type** является синонимом класса **Key**.

Функция-член	*Описание*
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент мультимножества.
`void clear();`	Удаляет все элементы из мультимножества.
`size_type count` `(const key_type &k) const;`	Возвращает количество экземпляров ключа *k* в мультимножестве.
`bool empty() const;`	Возвращает значение **true**, если вызывающее мультимножество пусто, и значение **false** — в противном случае
`iterator end();` `const_iterator end() const;`	Возвращает итератор, установленный на конец мультимножества.
`pair<iterator,iterator>` `equal_range(const key_type &k);`	Возвращает пару итераторов, установленных на первый и последний элементы мультимножества, содержащих указанный ключ.
`void erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`void erase(iterator start,` `iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`size_type erase` `(const key_type &k);`	Удаляет из мультимножества элементы, имеющие ключ со значением *k*.
`iterator find` `(const key_type &k) const;`	Возвращает итератор, установленный на заданный ключ. Если ключ не найден, возвращается итератор, ссылающийся на конец мультимножества.
`allocator_type` `get_allocator() const;`	Возвращает распределитель памяти для мультимножества
`iterator insert(iterator i,` `const value_type &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.

`template <class InIter>`
`void insert(InIter start,`
`InIter end);`
	Вставляет элементы последовательности, определенной итераторами *start* и *end*.

`iterator insert` `(const value_type &val);`	Вставляет значение *val* в вызывающее мультимножество.
`key_compare key_comp() const;`	Возвращает функтор, сравнивающий ключи.
`iterator lower_bound` `(const key_type &k) const;`	Возвращает итератор, установленный на первый элемент мультимножества, ключ которого равен или превышает значение *k*.
`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в мультимножестве.

Окончание табл.

Функция-член	Описание
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный на конец мультимножества.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начало мультимножества.
`size_type size() const;`	Возвращает текущее количество элементов мультимножества.
`void swap(map<Key, Comp, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем мультимножестве, и элементы из объекта *ob*.
`iterator upper_bound(const key_type &k) const;`	Возвращает итератор, установленный на первый элемент мультимножества, ключ которого превышает значение *k*.
`value_compare value_comp() const;`	Возвращает функтор, сравнивающий два значения.

Класс queue

Класс **queue** обеспечивает работу с односторонней очередью. Его шаблонная спецификация выглядит следующим образом.

```
template <class T, class Container = deque<T> > class queue
```

Здесь **T** обозначает тип данных, хранящихся в классе **queue**, а класс **Container** описывает тип контейнера, в котором хранится очередь. Этот класс содержит следующий конструктор.

```
explicit queue(const Container &cnt = Container());
```

Конструктор **queue()** создает пустую очередь. По умолчанию в качестве контейнера используется класс **deque**, однако элементы очереди организованы по принципу "первым вошел — первым вышел". В качестве контейнера можно также применять класс **list**. Контейнер содержит защищенный объект **c**, имеющий тип **Container**.

Кроме того, в классе **queue** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **queue** содержит такие функции-члены.

Функция-член	Описание
`value_type &back();` `const value_type &back() const;`	Возвращает ссылку на последний элемент очереди.
`bool empty() const;`	Возвращает значение **true**, если вызывающая очередь пуста, и **false** — в противном случае.
`value_type &front();` `const value_type &front() const;`	Возвращает итератор, установленный на первый элемент очереди.
`void pop()`	Удаляет первый элемент очереди.
`void push(const value_type &val);`	Добавляет элемент, имеющий значение *val*, в конец очереди.
`size_type size() const;`	Возвращает текущее количество элементов, находящихся в очереди.

Класс priority_queue

Класс **priority_queue** обеспечивает работу с односторонней очередью с приоритетами. Его шаблонная спецификация выглядит следующим образом.

```
template <class T, class Container = vector<T>,
          class Comp = less<Container::value_type > >
          class priority_queue
```

Здесь **T** обозначает тип данных, хранящихся в классе **queue**, а класс **Container** описывает тип контейнера, в котором хранится очередь. Этот класс содержит следующий конструктор.

```
explicit priority_queue(const Comp &cmpfn = Comp(),
                        Container &cnt = Container());

template <class InIter> priority_queue(InIter start, InIter end,
                        const Comp &cmpfn = Comp(),
                        Container &cnt = Container());
```

Первый конструктор **priority_queue()** создает пустую очередь с приоритетами. Второй конструктор создает очередь с приоритетами, содержащую элементы, заданные итераторами *start* и *end*. По умолчанию в качестве контейнера используется класс **vector**. В качестве контейнера можно также применять класс **deque**. Контейнер содержит защищенный объект **c**, имеющий тип **Container**.

Класс **priority_queue** содержит следующие функции-члены

Функция-член	*Описание*
bool empty() const;	Возвращает значение **true**, если вызывающая очередь пуста, и **false** — в противном случае.
void pop()	Удаляет первый элемент очереди.
void push(const T &*val*);	Добавляет элемент, имеющий значение *val*, в очередь с приоритетами.
size_type size() const;	Возвращает текущее количество элементов, находящихся в очереди.
const value_type &top() const;	Возвращает ссылку на элемент, имеющий наивысший приоритет. Сам элемент не удаляется.

Класс set

Класс **multiset** обеспечивает работу с множеством, в котором каждому значению может соответствовать лишь один ключ. Его шаблонная спецификация выглядит следующим образом

```
template <class Key, class Comp = less<Key>,
   class Allocator = allocator<Key> class set
```

Здесь **Key** означает тип ключа, а **Comp** представляет собой функцию, предназначенную для сравнения двух ключей. Этот класс содержит следующие конструкторы.

```
explicit set(const Comp &cmpfn = Comp(),
             const Allocator &a = Allocator());

set(const set<Key, Comp, Allocator> &ob);
```

```
template <class InIter> set(InIter start, InIter end,
   const Comp &cmpfn = Comp(),
   const Allocator &a = Allocator());
```

Первый конструктор создает пустое множество. Второй конструктор создает множество, содержащее элементы объекта *ob*. Третий конструктор создает множество, содержащее элементы, изменяющиеся в диапазоне от *start* до *end*. Параметр *cmpfn* задает функцию, определяющую порядок следования элементов в множестве.

Кроме того, в классе **set** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **set** содержит такие функции-члены.

Функция-член	*Описание*
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент множества.
`void clear();`	Удаляет все элементы из множества.
`size_type count` `(const key_type &k) const;`	Возвращает количество экземпляров ключа *k* в множестве.
`bool empty() const;`	Возвращает значение **true**, если вызывающее множество пусто, и **false** — в противном случае.
`const_iterator end() const;` `iterator end();`	Возвращает итератор, установленный на конец множества.

`pair<iterator,iterator>`
`equal_range(const key_type &k)`
`const;`
	Возвращает пару итераторов, установленных на первый и последний элементы множества, содержащих указанный ключ.

`void erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`void erase(iterator start,` `iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`size_type erase(const key_type &k);`	Удаляет из множества элементы, имеющие ключ со значением *k*.
`iterator find(const key_type &k);`	Возвращает итератор, установленный на заданный ключ. Если ключ не найден, возвращается итератор, ссылающийся на конец множества.
`allocator_type get_allocator()` `const;`	Возвращает распределитель памяти для множества.
`iterator insert(iterator i,` `const value_type &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.

`template <class InIter>`
`void insert(InIter start,`
`InIter end);`
	Вставляет элементы последовательности, определенной итераторами *start* и *end*.

`pair<iterator,bool>` `insert(const value_type &val);`	Вставляет значение *val* в вызывающее множество.
`key_compare key_comp() const;`	Возвращает функтор, сравнивающий ключи.
`iterator lower_bound` `(const key_type &k) const;`	Возвращает итератор, установленный на первый элемент множества, ключ которого равен или превышает значение *k*.

Окончание табл.

Функция-член	Описание
`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в множестве.
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный на конец множества.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в начало множества.
`size_type size() const;`	Возвращает текущее количество элементов множества
`void swap(set<Key, Comp,Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем множестве, и элементы из объекта *ob*.
`iterator upper_bound (const key_type &k)const;`	Возвращает итератор, установленный на первый элемент множества, ключ которого превышает значение *k*.
`value_compare value_comp() const;`	Возвращает функтор, сравнивающий два значения.

Класс stack

Класс **stack** обеспечивает работу со стеком. Его шаблонная спецификация выглядит следующим образом

```
template <class T, class Container = deque<T> > class stack
```

Здесь **T** обозначает тип данных, хранящихся в классе **stack**, а класс **Container** описывает тип контейнера, в котором хранится стек. Этот класс содержит следующий конструктор.

```
explicit stack(const Container &cnt = Container());
```

Конструктор **stack()** создает пустой стек. По умолчанию в качестве контейнера используется класс **deque**, однако элементы стека организованы по принципу "последним вошел — первым вышел" В качестве контейнера можно также применять классы **vector** или **list**. Контейнер содержит защищенный объект **c**, имеющий тип **Container**.

Кроме того, в классе **stack** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **stack** содержит такие функции-члены.

Функция-член	Описание
`bool empty() const;`	Возвращает значение **true**, если вызывающий стек пуст, и **false** — в противном случае.
`void pop()`	Удаляет вершину стека, который с технической точки зрения является последним элементом контейнера.
`void push(const value_type &val);`	Добавляет элемент, имеющий значение *val*, в конец стека. Последний элемент контейнера является вершиной стека.
`size_type size() const;`	Возвращает текущее количество элементов, находящихся в стеке.
`const value_type &top() const;`	Возвращает ссылку на вершину стека, т.е. на последний элемент контейнера. Сам элемент не удаляется.

Класс vector

Класс **vector** обеспечивает работу с динамическим массивом. Его шаблонная спецификация выглядит следующим образом.

```
template <class T, class Allocator = allocator<T> > class vector
```

Здесь **T** обозначает тип данных, хранящихся в классе **vector**, а класс **Allocator** задает способ распределения памяти. Этот класс содержит следующие конструкторы.

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type num, const T &val = T(),
   const Allocator &a = Allocator());

vector(const vector<T, Allocator> &ob);

template <class InIter> vector(InIter start, InIter end,
   const Allocator &a = Allocator());
```

Первый конструктор создает пустой вектор. Второй конструктор создает вектор, содержащий *num* элементов, имеющих значение *val*. Третий конструктор создает вектор, содержащий элементы объекта *ob*. Четвертый конструктор создает вектор, содержащий элементы, изменяющиеся в диапазоне от *start* до *end*.

Кроме того, в классе **vector** определены следующие операции.

```
==, <, <=, !=, >, >=
```

Класс **vector** содержит такие функции-члены.

Функция-член	*Описание*
template <class InIter> void assign(InIter *start*, **InIter** *end*);	Присваивает вектору последовательность элементов, определенную итераторами *start* и *end*.
void assign(size_type *num*, **const T &***val*);	Присваивает вектору *num* элементов, имеющих значение *val*.
reference at(size_t *i*); **const_reference at(size_t** *i*) **const;**	Возвращает ссылку на *i*-й элемент вектора.
reference back(); const_reference back() const;	Возвращает итератор, установленный на последний элемент вектора.
iterator begin(); const_iterator begin() const;	Возвращает итератор, установленный на первый элемент вектора.
size_type capacity() const;	Возвращает текущую емкость вектора. Она равна количеству элементов, которые можно хранить в векторе.
void clear();	Удаляет все элементы из вектора.
bool empty() const;	Возвращает значение **true**, если вызывающий вектор пуст, и **false** — в противном случае.
iterator end(); const_iterator end() const;	Возвращает итератор, установленный на конец вектора.

Функция-член	Описание
`iterator erase(iterator i);`	Удаляет элемент, на который ссылается итератор *i*. Возвращает итератор, установленный на следующий элемент.
`iterator erase(iterator start, iterator end);`	Удаляет элементы в диапазоне от итератора *start* до итератора *end*. Возвращает итератор, установленный на элемент, следующий за последним удаленным элементом.
`reference front();` `const_reference front() const;`	Возвращает итератор, установленный на первый элемент двусторонней очереди.
`allocator_type get_allocator() const;`	Возвращает распределитель памяти для вектора.
`iterator insert(iterator i, const T &val);`	Вставляет значение *val* непосредственно перед элементом, на который ссылается итератор *i*. Возвращает итератор, установленный на этот элемент.
`void insert(iterator i, size_type num, const T &val);`	Вставляет *num* копий значения *val* непосредственно перед элементом, на который ссылается итератор *i*.
`template <class InIter> void insert(iterator i, InIter start, InIter end);`	Вставляет элементы последовательности, определенной итераторами *start* и *end*, непосредственно перед элементом, на который ссылается итератор *i*.
`size_type max_size() const;`	Возвращает максимальное количество элементов, которые могут храниться в векторе
`reference operator[] (size_type i)const;` `const_reference operator[] (size_type i) const;`	Возвращает ссылку на *i*-й элемент.
`void pop_back();`	Удаляет последний элемент векторaa.
`void push_back(const T &val);`	Добавляет элемент, имеющий значение *val*, в конец вектора.
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный в начало вектор.
`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный в конец вектора.
`void reserve(size_type num);`	Устанавливает емкость вектора равной или превосходящей *num*.
`void resize(size_type num, T val=T());`	Меняет размер вектора на значение, заданное параметром *num*. Если вектор необходимо удлинить, элементы, имеющие значение *val*, добавляются в конец.
`size_type size() const;`	Возвращает текущее количество элементов, находящихся в векторе.
`void swap(vector<T, Allocator> &ob);`	Меняет местами элементы, хранящиеся в вызывающем векторе, и объекты *ob*.

Библиотека STL также содержит специализацию класса **vector** для булевых значений. Он обладает всеми функциональными возможностями вектора и добавляет еще две функции-члена.

`void flip();`	Инвертирует все биты, хранящиеся в векторе.
`static void swap(reference i,reference j);`	Меняет местами *i*-й и *j*-й элементы вектора.

Полный справочник по

Глава 34

Стандартные алгоритмы

В этой главе описываются алгоритмы, определенные в стандартной библиотеке шаблонов. Они оперируют с контейнерами с помощью итераторов и реализуются в виде шаблонных функций. Определения имен обобщенных типов, используемых стандартными алгоритмами, приведены ниже.

Обобщенное имя	*Смысл*
`BiIter`	Двунаправленный итератор
`ForIter`	Прямой итератор
`InIter`	Итератор ввода
`OutIter`	Итератор вывода
`RandIter`	Итератор произвольного доступа
`T`	Некий тип данных
`Size`	Некая разновидность целочисленного типа
`Func`	Некий тип функции
`Generator`	Функция, генерирующая объекты
`BinPred`	Бинарный предикат
`UnPred`	Унарный предикат
`Comp`	Функция сравнения

Алгоритм adjacent_find

```
template <class ForIter>
   ForIter adjacent_find(ForIter start, ForIter end);
template <class ForIter, class BinPred>
   ForIter adjacent_find(ForIter start, ForIter end, BinPred pfn);
```

Алгоритм **`adjacent_find()`** выполняет поиск одинаковых соседних элементов в последовательности, определенной итераторами *start* и *end*, и возвращает итератор, установленный на первый элемент. Если искомая пара не найдена, алгоритм возвращает итератор *end*. Первая версия алгоритма выполняет поиск эквивалентных элементов.

Вторая версия позволяет программисту самому указать метод определения совпадающих элементов.

Алгоритм binary_search

```
template <class ForIter, class T>
   bool binary_search(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
   bool binary_search(ForIter start, ForIter end, const T &val, Comp cmpfn);
```

Алгоритм **`binary_search()`** выполняет бинарный поиск элемента, имеющего значение *val*, в последовательности, определенной итераторами *start* и *end*. Если искомый элемент найден, алгоритм возвращает значение **`true`**, в противном случае он возвращает значение **`false`**. Первая версия алгоритма выполняет поиск, проверяя элементы на равенство. Вторая версия позволяет программисту самому задать функцию сравнения.

Алгоритм copy

```
template <class InIter, class OutIter>
   Outer copy(InIter start, InIter end, OutIter result);
```

Алгоритм **copy()** копирует последовательность, определенную итераторами *start* и *end*, в последовательность, на которую ссылается итератор **result**. Алгоритм возвращает итератор, установленный на результирующую последовательность. Копируемый диапазон не должен перекрываться с последовательностью *result*.

Алгоритм copy_backward

```
template <class BiIter1, class BiIter2>
   BiIter2 copy_backward(BiIter1 start, BiIter1 end, BiIter2 result);
```

Алгоритм **copy_backward()** совпадает с алгоритмом **copy()**. Единственное отличие заключается в том, что результирующая последовательность записывается в обратном порядке.

Алгоритм count

```
template <class InIter, class T>
   ptrdiff_t count(InIter start, InIter end, const T &val);
```

Алгоритм **count()** возвращает количество элементов, имеющих значение *val*, в последовательности, определенной итераторами *start* и *end*.

Алгоритм count_if

```
template <class InIter, class T>
   ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

Алгоритм **count_if()** возвращает количество элементов, соответствующих унарному предикату *pfn*, в последовательности, определенной итераторами *start* и *end*. Тип **prtdiff_t** представляет собой разновидность целого числа.

Алгоритм equal

```
template <class InIter1, class InIter2>
   bool equal(InIter1 start1, InIter1 end1, InIter2 start2);
template <class InIter1, class InIter2, class BinPred>
   bool equal(InIter1 start1, InIter1 end1, InIter2 start2, Binpred pfn);
```

Алгоритм **equal()** проверяет, совпадает ли диапазон, ограниченный итераторами *start1* и *end1*, с последовательностью, на которую ссылается итератор *start2*. Если диапазоны совпадают, возвращается значение **true**, если нет — **false**.

Вторая версия позволяет программисту самому задать бинарный предикат, определяющий условие эквивалентности двух элементов.

Алгоритм equal_range

```
template <class ForIter, class T>
   pair<ForIter, ForIter> equal_range(ForIter start,
   ForIter end, const T &val);
template <class ForIter, class T, class Comp>
   pair<ForIter, ForIter> equal_range(ForIter start,
   ForIter end, const T &val, Comp cmpfn);
```

Алгоритм **equal_range()** возвращает диапазон, допускающий вставку элементов без нарушения порядка. Область поиска такого диапазона ограничена итераторами *start*

и *end*. Значение, подлежащее вставке, задается параметром *val*. Вторая версия позволяет программисту самому задать функцию сравнения, определяющую критерий поиска.

Шаблонный класс **pair** является вспомогательным. Он позволяет хранить пару объектов в полях **first** и **second**.

Алгоритмы fill и fill_n

```
template <class ForIter, class T>
   void fill(ForIter start, ForIter end, const T &val);
template <class OutIter, class Size, class T>
   void fill_n(OutIter start, Size num, const T &val);
```

Алгоритмы **fill()** и **fill_n()** заполняют диапазон значением *val*. Диапазон, заполняемый функцией **fill()**, ограничен итераторами *start* и *end*. Начало диапазона, заполняемого функцией **fill_n()**, задается итератором *start*. Он содержит *num* элементов

Алгоритм find

```
template <class InIter, class T>
   InIter find(InIter start, InIter end, const T &val);
```

Алгоритм **find()** выполняет поиск элемента, имеющего значение *val*, в последовательности, ограниченной итераторами *start* и *end*. Если искомый элемент найден, алгоритм возвращает итератор, установленный на его первое вхождение. В противном случае он возвращает итератор *end*.

Алгоритм find_end

```
template <class ForIter1, class ForIter2>
   ForIter1 find_end(ForIter1 start1, ForIter1 end1,
                     ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
   ForIter1 find_end(ForIter1 start1, ForIter1 end1,
                     ForIter2 start2, ForIter2 end2,
                     BinPred pfn);
```

Алгоритм **find_end()** выполняет поиск последнего вхождения подпоследовательности, ограниченной итераторами *start2* и *end2*, в диапазон, определенный итераторами *start1* и *end1*. Если подпоследовательность найдена, алгоритм возвращает итератор, установленный на ее первый элемент. В противном случае он возвращает итератор *end1*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать бинарный предикат, определяющий условия совпадения.

Алгоритм find_first_of

```
template <class ForIter1, class ForIter2>
   ForIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                          ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
   ForIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                          ForIter2 start2, ForIter2 end2,
                          BinPred pfn);
```

Алгоритм **find_first_of()** выполняет поиск первого элемента подпоследовательности, ограниченной итераторами *start2* и *end2*, совпадающего с каким-либо эле-

ментом диапазона, определенного итераторами *start1* и *end1*. Если такой элемент не найден, алгоритм возвращает итератор *end1*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать бинарный предикат, определяющий условия совпадения.

Алгоритм find_if

```
template <class InIter, class UnPred>
   InIter find_if(InIter start, InIter end, UnPred pfn);
```

Алгоритм **find_if()** выполняет поиск элемента, соответствующего унарному предикату *pfn*, в диапазоне, ограниченном итераторами *start1* и *end1*. Если такой элемент найден, алгоритм возвращает итератор, установленный на его первое вхождение, в противном случае он возвращает итератор *end1*.

Алгоритм for_each

```
template <class InIter, class Func>
   Func for_each(InIter start, InIter end, Func fn);
```

Алгоритм **for_each()** применяет функцию *fn* к диапазону элементов, ограниченному итераторами *start* и *end*. Алгоритм возвращает функцию *fn*.

Алгоритмы generate и generate_n

```
template <class ForIter, class Generator>
   void generate(ForIter start, ForIter end, Generator fngen);
template <class ForIter, class Size, class Generator>
   void generate_n(OutIter start, Size num, Generator fngen);
```

Алгоритмы **generate()** и **generate_n()** присваивают элементам диапазона значение, возвращаемое функцией *fngen*. Для функции **generate()** диапазон элементов ограничен итераторами *start* и *end*. Начало диапазона, заполняемого функцией **generate_n()**, задается итератором *start*. Он содержит *num* элементов. Функция, генерирующая значения элементов, задается параметром *fngen*. Она не имеет параметров.

Алгоритм includes

```
template <class InIter1, class InIter2>
   bool includes(InIter1 start1, InIter1 end1,
                 InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
   bool includes(InIter1 start1, InIter1 end1,
                 InIter2 start2, InIter2 end2, Comp cmpfn);
```

Алгоритм **includes()** определяет, содержит ли последовательность, ограниченная итераторами *start1* и *end1*, элементы последовательности, определенной итераторами *start2* и *end2*. Если ответ утвердительный, алгоритм возвращает значение **true**, в противном случае возвращается значение **false**.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм inplace_merge

```
template <class BiIter>
   void inplace_merge(BiIter start, BiIter mid, BiIter end);
```

```
template <class BiIter, class Comp>
   void inplace_merge(BiIter start, BiIter mid, BiIter end, Comp cmpfn);
```

Алгоритм **inplace_merge()** объединяет диапазон, ограниченный итераторами *start* и *end*, с диапазоном, определенным итераторами *mid* и *end*. При этом оба диапазона принадлежат одной и той же последовательности и должны быть упорядочены по возрастанию. Результирующая последовательность является упорядоченной по возрастанию.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм iter_swap

```
template <class ForIter1, class ForIter2>
   void iter_swap(ForIter1 i, ForIter2 j);
```

Алгоритм **iter_swap()** меняет местами значения, адресуемые двумя итераторами.

Алгоритм lexicographical_compare

```
template <class InIter1, class InIter2>
   bool lexicographical_compare(InIter1 start1, InIter1 end1,
                                InIter2 start2, InIter2 end2);
template <class InIter1, class InIter2, class Comp>
   bool lexicographical_compare(InIter1 start1, InIter1 end1,
                                InIter2 start2, InIter2 end2,
                                Comp cmpfn);
```

Алгоритм **lexicographical_compare()** выполняет лексикографическое сравнение последовательности, ограниченной итераторами *start1* и *end1*, с последовательностью, определенной итераторами *start2* и *end2*. Если первая последовательность предшествует второй по алфавиту, алгоритм возвращает значение **true**, в противном случае возвращается значение **false**.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм lower_bound

```
template <class ForIter, class T>
   ForIter lower_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
   ForIter lower_bound(ForIter start, ForIter end,
                       const T &val, Comp cmpfn);
```

Алгоритм **lower_bound()** находит первый элемент последовательности, ограниченной итераторами *start* и *end*, значение которого не превышает значение параметра *val*. Возвращается итератор, установленный на искомый элемент.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения, определяющую критерий поиска.

Алгоритм make_heap

```
template <class RandIter>
   void make_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
   void make_heap(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **make_heap()** создает кучу из последовательности, ограниченной итераторами *start* и *end*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения, определяющую критерий поиска.

Алгоритм max

```
template <class T>
  const T &max(const T &i, const T &j);
template <class T, class Comp>
  const T &max(const T &i, const T &j, Comp cmpfn);
```

Алгоритм **max()** возвращает максимальное из двух значений.

Вторая версия позволяет программисту самому задать функцию сравнения

Алгоритм max_element

```
template <class ForIter>
  ForIter max_element(ForIter start, ForIter last);
template <class ForIter, class Comp >
  ForIter max_element(ForIter start, ForIter last, Comp cmpfn);
```

Алгоритм **max_element()** возвращает итератор, установленный на максимальный элемент в диапазоне, ограниченном итераторами *start* и *last*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм merge

```
template <class InIter1, class InIter2, class OutIter>
   bool includes(InIter1 start1, InIter1 end1,
                 InIter2 start2, InIter2 end2,
                 OutIter result);
template <class InIter1, class InIter2, class Outer, class Comp>
   bool includes(InIter1 start1, InIter1 end1,
                 InIter2 start2, InIter2 end2,
                 OutIter result, Comp cmpfn);
```

Алгоритм **merge()** объединяет две упорядоченные последовательности, помещая результат в третью последовательность. Объединяемые последовательности ограничены итераторами **start1** и **end1**, а также **start2** и **end2** соответственно. Результат записывается в последовательность, адресуемую итератором *result*. Алгоритм возвращает итератор, установленный на конец результирующей последовательности.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм min

```
template <class T>
  const T &min(const T &i, const T &j);
template <class T, class Comp>
  const T &min(const T &i, const T &j, Comp cmpfn);
```

Алгоритм **min()** возвращает минимальное из двух значений.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм min_element

```
template <class ForIter>
  ForIter min_element(ForIter start, ForIter last);
template <class ForIter, class Comp>
  ForIter min_element(ForIter start, ForIter last, Comp cmpfn);
```

Алгоритм **min_element()** возвращает итератор, установленный на минимальный элемент в диапазоне, ограниченном итераторами *start* и *last*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм mismatch

```
template <class InIter1, class InIter2>
  pair<InIter1, inIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2);
template <class InIter1, class InIter2, class BinPred>
  pair<InIter1, inIter2> mismatch(InIter1 start1, InIter1 end1,
                                  InIter2 start2, BinPred pfn);
```

Алгоритм **mismatch()** выполняет поиск первого несовпадения между двумя последовательностями. Возвращаются итераторы, установленные на первые несовпадающие элементы. Если такие элементы не найдены, алгоритм возвращает итераторы, установленные на последние элементы последовательности.

Вторая версия алгоритма позволяет программисту самостоятельно задавать бинарный предикат, определяющий условия совпадения.

Шаблонный класс **pair** является вспомогательным. Он позволяет хранить пару объектов в полях **first** и **second**.

Алгоритм next_permutation

```
template <class BiIter>
   bool next_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
   bool next_permutation(BiIter start, BiIter end, Comp cmpfn);
```

Алгоритм **next_permutation()** создает следующую перестановку, состоящую из элементов последовательности. Перестановки генерируются на основе упорядоченной последовательности, которая считается первой перестановкой. Если следующей перестановки не существует, алгоритм **next_permutation()** упорядочивает последовательность, представляющую собой первую перестановку, и возвращает значение **false**. В противном случае возвращается значение **true**.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм nth_element

```
template <class RandIter>
  void nth_element(RandIter start, RandIter element,
                   RandIter end);
template <class RandIter, class Comp>
 void nth_element(RandIter start, RandIter element,
                   RandIter end, Comp cmpfn);
```

Алгоритм **nth_element()** упорядочивает последовательность, ограниченную итераторами *start* и *end*, чтобы элементы, значения которых меньше параметра *element*, предшествовали ему, а элементы, значения которых больше, — следовали за ним.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм partial_sort

```
template <class RandIter>
  void partial_sort(RandIter start, RandIter mid,
                    RandIter end);
template <class RandIter, class Comp>
  void partial_sort(RandIter start, RandIter mid,
                    RandIter end, Comp cmpfn);
```

Алгоритм **partial_sort()** упорядочивает последовательность, ограниченную итераторами *start* и *end*. Однако после выполнения алгоритма упорядоченными оказываются лишь элементы из диапазона, ограниченного итераторами *start* и *mid*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм partial_sort_copy

```
template <class InIter, class RandIter>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end);
template <class InIter, class RandIter, class Comp>
  RandIter partial_sort_copy(InIter start, InIter end,
                             RandIter res_start, RandIter res_end,
                             Comp cmpfn);
```

Алгоритм **partial_sort_copy()** упорядочивает последовательность, ограниченную итераторами *start* и *end*, а затем копирует в результирующую последовательность, определенную итераторами *res_start* и *res_end*, столько элементов, сколько она может вместить. Он возвращает итератор, установленный на элемент результирующей последовательности, который был скопирован последним.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм partition

```
template <class BiIter, class UnPred>
  BiIter partition(BiIter start, BiIter end, UnPred pfn);
```

Алгоритм **partition()** упорядочивает последовательность, ограниченную итераторами *start* и *end*, так, чтобы все элементы, для которых унарный предикат *pfn* является истинным, предшествовали тем, для которых этот предикат возвращает ложное значение. Алгоритм возвращает итератор, установленный в начало элементов, для которых предикат является ложным.

Алгоритм pop_heap

```
template <class RandIter>
  void pop_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void pop_heap(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **pop_heap()** меняет местами первый и предпоследний элементы и перестраивает кучу.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм prev_permutation

```
template <class BiIter>
   bool prev_permutation(BiIter start, BiIter end);
template <class BiIter, class Comp>
   bool prev_permutation(BiIter start, BiIter end, Comp cmpfn);
```

Алгоритм **prev_permutation()** воссоздает предыдущую перестановку, состоящую из элементов последовательности. Перестановки генерируются на основе упорядоченной последовательности, которая считается первой перестановкой. Если следующей перестановки не существует, алгоритм **next_permutation()** упорядочивает последовательность, представляющую собой последнюю перестановку, и возвращает значение **false**. В противном случае возвращается значение **true**.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм push_heap

```
template <class RandIter>
  void push_heap(RandIter start, RandIter end);
template <class RandIter, class Comp >
  void push_heap(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **push_heap()** заталкивает элемент в конец кучи. Предполагается, что диапазон, ограниченный итераторами *start* и *end*, представляет собой корректную кучу.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения.

Алгоритм random_shuffle

```
template <class RandIter>
  void random_shuffle(RandIter start, RandIter end);
template <class RandIter, class Generator>
  void random_shuffle(RandIter start, RandIter end,
                      Generator  rand_gen);
```

Алгоритм **random_shuffle()** перетасовывает последовательность, ограниченную итераторами *start* и *end*.

Вторая версия позволяет программисту самостоятельно задавать генератор случайных чисел. Эта функция должна иметь следующий вид и возвращать случайное число, лежащее между числами 0 и *num*.

```
rand_gen(num);
```

Алгоритмы remove, remove_if, remove_copy и remove_copy_if

```
template <class ForIter, class T>
   ForIter remove(ForIter start, ForIter end, const T &val);
template <class ForIter, class UnPred>
   ForIter remove_if(ForIter start, ForIter end, UnPred pfn);
template <class InIter, class OutIter, class T>
   OutIter remove_copy(InIter start, InIter end,
                       OutIter result, const T &val);
template <class InIter, class OutIter, class UnPred>
   OutIter remove_copy_if(InIter start, InIter end,
                          OutIter result, UnPred pfn);
```

Алгоритм **remove()** удаляет из указанной последовательности элементы, имеющие значение *val*. Возвращается итератор, установленный на последний из оставшихся элементов.

Алгоритм **remove_if()** удаляет из указанной последовательности элементы, для которых предикат *pfn* является истинным. Возвращается итератор, установленный на последний из оставшихся элементов.

Алгоритм **remove_copy()** копирует из указанной последовательности элементы, имеющие значение *val*, и записывает результат в последовательность, адресуемую итератором **result**. Возвращается итератор, установленный на последний из оставшихся элементов.

Алгоритм **remove_copy_if()** копирует из указанной последовательности элементы, для которых предикат *pfn* является истинным, и записывает результат в последовательность, адресуемую итератором **result**. Возвращается итератор, установленный на последний из оставшихся элементов.

Вторая версия позволяет программисту самому задать функцию сравнения, определяющую критерий поиска.

Алгоритмы replace, replace_copy, replace_if и replace_copy_if

```
template <class ForIter, class T>
   void replace(ForIter start, ForIter end,
                const T &old, const T &new);
template <class ForIter, class UnPred, class T>
   void replace_if(ForIter start, ForIter end,
                   UnPred pfn, const T &new);
template <class InIter, class OutIter, class T>
   OutIter replace_copy(InIter start, InIter end,
                        OutIter result, const T &old,
                        const T &new);
template <class InIter, class OutIter, class UnPred, class T>
   OutIter replace_copy_if(InIter start, InIter end,
                           OutIter result, UnPred pfn,
                           const T &new);
```

Алгоритм **replace()** заменяет элементы заданного диапазона, имеющие значение *old*, элементами, имеющими значение *val*.

Алгоритм **replace_if()** заменяет элементы заданного диапазона, соответствующие предикату *pfn*, элементами, имеющими значение *new*.

Алгоритм **replace_copy_if()** копирует элементы заданного диапазона в последовательность, адресуемую итератором *result*, заменяя элементы, имеющие значение *old*, элементами, имеющими значение *new*. Исходный диапазон не изменяется. Возвращается итератор, установленный на конец последовательности *result*.

Алгоритм **replace_copy()** копирует элементы заданного диапазона в последовательность, адресуемую итератором *result*, заменяя элементы, соответствующие предикату *pfn*, элементами, имеющими значение *new*. Исходный диапазон не изменяется. Возвращается итератор, установленный на конец последовательности *result*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения, определяющую критерий поиска.

Алгоритмы reverse и reverse_copy

```
template <class BiIter>
  void reverse(BiIter start, BiIter end);
template <class BiIter, class OutIter>
  OutIter reverse_copy(BiIter start, BiIter end, OutIter result);
```

Алгоритм **reverse()** меняет порядок следования элементов диапазона, ограниченного итераторами *start* и *end*.

Алгоритм **reverse_copy()** меняет порядок следования элементов диапазона, ограниченного итераторами *start* и *end*, и копирует результат в последовательность, адресуемую итератором *result*. Алгоритм возвращает итератор, установленный на конец результирующей последовательности.

Алгоритмы rotate и rotate_copy

```
template <class ForIter>
  void rotate(ForIter start, ForIter mid, ForIter end);
template <class ForIter, class OutIter>
  OutIter rotate_copy(ForIter start, ForIter mid,
                      ForIter end, OutIter result);
```

Алгоритм **rotate()** выполняет левое вращение элементов диапазона, ограниченного итераторами *start* и *end*, так что элемент, заданный итератором *mid*, становится первым.

Алгоритм **rotate_copy()** выполняет левое вращение элементов диапазона, ограниченного итераторами *start* и *end*, и копирует результат в последовательность, адресуемую итератором *result*. Алгоритм возвращает итератор, установленный на конец результирующей последовательности.

Алгоритм search

```
template <class ForIter1, class ForIter2>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  ForIter1 search(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2,
                  BinPred pfn);
```

Алгоритм **search()** выполняет поиск подпоследовательности, ограниченной итераторами *start2* и *end2*, в диапазоне, определенном итераторами *start1* и *end1*. Если такая подпоследовательность не найдена, алгоритм возвращает итератор *end1*. В случае успеха алгоритм возвращает итератор, установленный в начало обнаруженной подпоследовательности.

Вторая версия алгоритма позволяет программисту самостоятельно задавать бинарный предикат, определяющий условия совпадения элементов.

Алгоритм search_n

```
template <class ForIter, class Size, class T>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val);
template <class ForIter, class Size, class T, class BinPred>
  ForIter search_n(ForIter start, ForIter end,
                   Size num, const T &val, BinPred pfn);
```

Алгоритм **search_n()** выполняет поиск последовательности, состоящей из *num* элементов, имеющих значение *val*, внутри последовательности, ограниченной итераторами *start* и *end*. Если такая подпоследовательность найдена, алгоритм возвращает итератор, установленный в ее начало. В противном случае алгоритм возвращает итератор *end*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать бинарный предикат, определяющий условия совпадения элементов.

Алгоритм set_difference

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_difference(InIter1 start1, InIter1 end1,
                         InIter2 start2, InIter2 end2,
                         OutIter result);
template <class InIter1, class InIter2,
          class OutIter, class Comp>
  OutIter set_difference(InIter1 start1, InIter1 end1,
                         InIter2 start2, InIter2 end2,
                         OutIter result, Comp cmpfn);
```

Алгоритм **set_difference()** создает последовательность, содержащую разность двух упорядоченных множеств, ограниченных итераторами *start1*, *end1* и *start2*, *end2* соответственно. Иначе говоря, множество, ограниченное итераторами *start2* и *end2*, вычитается из множества, определенного итераторами *start1* и *end1*. В результате возникает упорядоченное множество, которое записывается в последовательность, на которую ссылается итератор *result*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм set_intersection

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_intersection(InIter1 start1, InIter1 end1,
                           InIter2 start2, InIter2 end2,
                           OutIter result);
template <class InIter1, class InIter2,
          class OutIter, class Comp>
  OutIter set_intersection(InIter1 start1, InIter1 end1,
                           InIter2 start2, InIter2 end2,
                           OutIter result, Comp cmpfn);
```

Алгоритм **set_intersection()** создает последовательность, содержащую пересечение двух упорядоченных множеств, ограниченных итераторами *start1*, *end1* и *start2*, *end2* соответственно. Иначе говоря, возникает упорядоченное множество, состоящее из элементов, принадлежащих обоим параметрам одновременно, которое записывается в последовательность, адресуемую итератором *result*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм set_symmetric_difference

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
                                   InIter2 start2, InIter2 end2,
                                   OutIter result);
template <class InIter1, class InIter2,
          class OutIter, class Comp>
```

```
OutIter set_symmetric_difference(InIter1 start1, InIter1 end1,
                                 InIter2 start2, InIter2 end2,
                                 OutIter result, Comp cmpfn);
```

Алгоритм **set_symmetric_difference()** создает последовательность, содержащую симметричную разность двух упорядоченных множеств, ограниченных итераторами *start1*, *end1* и *start2*, *end2* соответственно. Иначе говоря, образуется последовательность, состоящая лишь из элементов, которые не принадлежат обоим множествам одновременно. Результирующая последовательность является упорядоченной и хранится в объекте, на который ссылается итератор *result*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм set_union

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_union(InIter1 start1, InIter1 end1,
                    InIter2 start2, InIter2 end2,
                    OutIter result);
template <class InIter1, class InIter2,
          class OutIter , class Comp>
  OutIter set_union(InIter1 start1, InIter1 end1,
                    InIter2 start2, InIter2 end2,
                    OutIter result, Comp cmpfn);
```

Алгоритм **set_union()** создает последовательность, содержащую объединение двух упорядоченных множеств, ограниченных итераторами *start1*, *end1* и *start2*, *end2* соответственно. Иначе говоря, возникает упорядоченное множество, состоящее из всех элементов обоих параметров, которое записывается в последовательность, адресуемую итератором *result*.

Вторая версия алгоритма позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм sort

```
template <class RandIter>
  void sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void sort(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **sort()** упорядочивает последовательность, ограниченную итераторами *start* и *end*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм sort_heap

```
template <class RandIter>
  void sort_heap(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void sort_heap(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **sort_heap()** упорядочивает кучу в диапазоне, ограниченном итераторами *start* и *end*.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм stable_partition

```
template <class BiIter, class UnPred>
   BiIter stable_partition(BiIter start, BiIter end, UnPred pfn);
```

Алгоритм **stable_partition()** упорядочивает последовательность, ограниченную итераторами *start* и *end*, так, чтобы все элементы, для которых унарный предикат *pfn* является истинным, предшествовали тем, для которых этот предикат возвращает ложное значение. Разбиение последовательности является устойчивым, т.е. при сортировке сохраняется относительная упорядоченность последовательности. Алгоритм возвращает итератор, установленный в начало элементов, для которых предикат является ложным.

Алгоритм stable_sort

```
template <class RandIter>
  void stable_sort(RandIter start, RandIter end);
template <class RandIter, class Comp>
  void stable_sort(RandIter start, RandIter end, Comp cmpfn);
```

Алгоритм **stable_sort()** упорядочивает последовательность, ограниченную итераторами *start* и *end*, не меняя местами эквивалентные элементы.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения элементов.

Алгоритм swap

```
template <class T>
   void swap(T &i, T &j);
```

Алгоритм **swap()** меняет местами значения, адресуемые ссылками *i* и *j*.

Алгоритм swap_ranges

```
template <class ForIter1, class ForIter2>
   ForIter2 swap_ranges(ForIter1 start1, ForIter1 end1,
                        ForIter2 start2);
```

Алгоритм **swap()** меняет местами элементы диапазона, ограниченного итераторами *start1* и *end1*, и последовательности, заданной итератором *start2*.

Алгоритм transform

```
template <class InIter, class OutIter, class Func>
   OutIter transform(InIter start, InIter end,
                     OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func>
   OutIter transform(InIter1 start1, InIter1 end1,
                     InIter2 start2, OutIter result,
                     Func binaryfunc);
```

Алгоритм **transform()** применяет функцию к диапазону элементов и записывает результат в последовательность *result*. В первом варианте диапазон ограничен итерато-

рами *start* и *end*. Функция задается параметром *unaryfunc*. Она получает значение элемента и возвращает результат его преобразования.

Во втором варианте преобразование представляет собой бинарную операторную функцию, получающую в качестве первого параметра элемент первой последовательности, а в качестве второго параметра — элемент второй последовательности.

В обоих версиях возвращается итератор, установленный на конец результирующей последовательности.

Алгоритм unique и unique_copy

```
template <class ForIter>
   ForIter unique(ForIter start, ForIter end);
template <class ForIter, class BinPred>
   ForIter unique(ForIter start, ForIter end, BinPred pfn);
template <class ForIter, class OutIter>
   OutIter unique_copy(ForIter start, ForIter end, OutIter result);
template <class ForIter, class OutIter, class BinPred>
   OutIter unique_copy(ForIter start, ForIter end,
                       OutIter result, BinPred pfn);
```

Алгоритм **unique()** исключает дубликаты из указанного диапазона. Второй вариант алгоритма позволяет программисту самостоятельно определить бинарный предикат, позволяющий распознать эквивалентные элементы. Алгоритм **unique()** возвращает итератор, установленный на конец диапазона.

Алгоритм **unique_copy()** копирует диапазон, ограниченный итераторами *start1* и *end1*, исключая дубликаты. Результат записывается в последовательность *result*. Вторая версия алгоритма позволяет программисту самостоятельно определить бинарный предикат, позволяющий распознать эквивалентные элементы. Алгоритм **unique_copy()** возвращает итератор, установленный на конец диапазона.

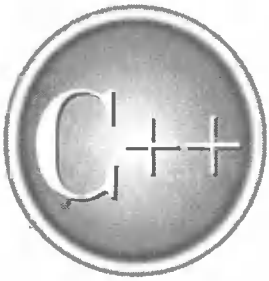
Алгоритм upper_bound

```
template <class ForIter, class T>
   ForIter upper_bound(ForIter start, ForIter end, const T &val);
template <class ForIter, class T, class Comp>
   ForIter upper_bound(ForIter start, ForIter end,
                       const T &val, Comp cmpfn);
```

Алгоритм **upper_bound()** находит последний элемент последовательности, ограниченной итераторами *start* и *end*, значение которого не превышает значение параметра *val*. Возвращается итератор, установленный на искомый элемент.

Вторая версия позволяет программисту самостоятельно задавать функцию сравнения, определяющую критерий поиска.

Полный справочник по



Глава 35

Стандартные итераторы, распределители памяти и функторы

В этой главе описываются классы и функции, обеспечивающие работу с итераторами, распределителями памяти и функторами. Эти компоненты являются частью стандартной библиотеки шаблонов. Следует заметить, что эти функции можно применять и для решения других задач.

Итераторы

Если контейнеры и алгоритмы образуют фундамент стандартной библиотеки шаблонов, то итераторы представляют собой цемент, связывающий их в одно целое. *Итератор* (iterator) — это обобщение (точнее, абстракция) указателя. Итераторы функционируют подобно указателям и реализуют стандартные операции над ними. Они позволяют перемещаться по содержимому контейнера так же, как указатель перемещается по элементам массива.

Стандарт языка C++ предусматривает набор классов и функций, обеспечивающих работу с итераторами. Однако в подавляющем большинстве задач программирования, требующих применения стандартной библиотеки шаблонов, эти классы и функции непосредственно не используются. Вместо них применяются итераторы, связанные с разнообразными контейнерами и выполняющие в них роль указателей. Тем не менее программист должен хорошо разбираться в принципах организации классов, реализующих итераторы, и понимать, как они устроены. Например, может возникнуть необходимость разработать свой собственный итератор, предназначенный для конкретной ситуации. Кроме того, итераторы могут оказаться полезными при разработке независимых библиотек.

Итераторы используют заголовок `<iterator>`.

Основные типы итераторов

Существуют пять видов итераторов.

Итератор	*Вид доступа*
Итератор произвольного доступа	Хранит и извлекает значения. Обеспечивает произвольный доступ к элементам.
Двунаправленный итератор	Хранит и извлекает значения. Перемещается вперед и назад.
Прямой итератор	Хранит и извлекает значения. Перемещается только вперед.
Итератор ввода	Извлекает, но не хранит элементы. Перемещается только вперед.
Итератор вывода	Хранит, но не извлекает значения. Перемещается только вперед.

Как правило, итераторы, обладающие более широкими возможностями, можно использовать вместо более слабых итераторов. Например, вместо итератора ввода можно применять прямой итератор.

В библиотеке STL существуют также *обратные итераторы* (reverse iterator). Они могут быть либо двунаправленными, либо итераторами произвольного доступа. В обоих случаях они должны иметь возможность перемещаться по контейнеру в обратном направлении. Таким образом, если обратный итератор ссылается на конец последовательности, его увеличение приведет к перемещению на предыдущий элемент.

Кроме них существуют еще два вида итераторов. Потоковые итераторы позволяют перемещаться по потоку. Итераторы вставки упрощают вставку элементов в контейнер.

Все итераторы должны поддерживать операции над соответствующими указателями. Например, класс итераторов ввода должен предусматривать операции `->`, `++`, `==` и `!=`. Кроме того, для присвоения значений нельзя применять оператор "`*`". В отличие от итераторов ввода, итератор произвольного доступа должен поддерживать операции `->`, `+`, `++`, `-`, `--`, `*`, `<`, `>`, `<=`, `>=`, `-=`, `+=`, `==` , `!=` и `[]`.

Классы низкоуровневых итераторов

В заголовке **<iostream>** определено несколько классов, обеспечивающих поддержку итераторов. Как указывалось в главе 24, каждый стандартный контейнер определяет свой собственный тип итератора **iterator** с помощью оператора **typedef**. Таким образом, используя стандартные контейнеры, программисты обычно не применяют низкоуровневые классы итераторов. Однако классы, описанные ниже, могут пригодиться программистам при разработке своих собственных итераторов.

В некоторых классах итераторов используется тип **ptrdiff_t**. Этот тип позволяет представить разность между двумя указателями.

Класс iterator

Класс **iterator** является базовым для всех итераторов. Вот как он выглядит.

```
template <class Cat, class T, class Dist = ptrdiff_t,
  class Pointer = T *, class Ref = T &>
struct iterator {
   typedef T value_type;
   typedef Dist difference_type;
   typedef Pointer pointer;
   typedef Ref reference;
   typedef Cat iterator_category;
}
```

Здесь тип **difference_type** позволяет хранить разность между двумя адресами, тип **value_type** описывает значения, над которыми выполняются операции, тип **pointer** задает вид указателя на значение, тип **reference** определяет вид ссылки на значение, а тип **iterator_category** задает тип итератора (т.е. итератор ввода, итератор произвольного доступа и т.д.).

Классы разделяются на следующие категории.

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag: public input_iterator_tag{};
struct bidirectional_iterator_tag: public
    forward_iterator_tag{};
struct random_access_iterator_tag: public
    bidirectional_iterator_tag{};
```

Класс iterator_traits

Класс **iterator_traits** обладает удобными возможностями для представления разнообразных типов, определенных итератором. Он формулируется следующим образом.

```
template<class Iterator> struct iterator_traits {
  typedef Iterator::difference_type difference_type;
  typedef Iterator::value_type value_type;
  typedef Iterator::pointer pointer;
  typedef Iterator::reference reference;
  typedef Iterator::iterator_category iterator_category;
}
```

Встроенные итераторы

Заголовок **<iterator>** содержит несколько встроенных итераторов, которые можно применять как непосредственно, так и для создания других итераторов. Эти итераторы перечислены в табл. 35.1. Обратите внимание на то, что в заголовке предусмот-

рены четыре потоковых итератора. Они позволяют алгоритмам манипулировать потоками. Кроме того, заслуживают внимания итераторы ввода. Если эти итераторы используются в операторе присваивания, они вставляют элементы в последовательность, а не перезаписывают ее содержимое.

Таблица 35.1. Встроенные итераторы

Класс	*Описание*
insert_iterator	Итератор вывода, выполняющий вставку в произвольное место контейнера
back_insert_iterator	Итератор вывода, выполняющий вставку в конец контейнера
front_insert_iterator	Итератор вывода, выполняющий вставку в начало контейнера
reverse_iterator	Обратный, двунаправленный или итератор произвольного доступа
istream_iterator	Потоковый итератор ввода
istreambuf_iterator	Потоковый итератор буферизованного ввода
ostream_iterator	Потоковый итератор вывода
ostreambuf_iterator	Потоковый итератор буфвризованного ввода

Рассмотрим каждый из перечисленных типов.

Класс insert_iterator

Класс **insert_iterator** поддерживает работу итераторов вывода, вставляющих объекты в контейнер. Его шаблонное определение выглядит следующим образом.

```
template <class Cont> class insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Здесь параметр **Cont** — это тип контейнера, для которого предназначен итератор. Класс **insert_iterator** имеет следующий конструктор.

```
insert_iterator(Cont &cnt, typename Cont::iterator itr);
```

Здесь параметр *cnt* представляет собой контейнер, для которого предназначен итератор, а параметр *itr* — это итератор вставки элементов в контейнер, который используется для инициализации объекта класса **insert_iterator**.

В классе **insert_iterator** определены следующие операторы: "=", "*", "++". Указатель на контейнер хранится в защищенной переменной **container**. Итератор контейнера хранится в защищенной переменной **iter**.

Кроме того, в классе определена функция **inserter()**, создающая объект класса **insert_iterator**. Ее спецификация представлена ниже.

```
template <class Cont, class Iterator> insert_iterator<Cont>
  inserter(Cont &cnt, Iterator itr);
```

Итераторы вставки предназначены для вставки элементов в контейнер, а не для их замены. Чтобы понять принцип работы итератора вставки, рассмотрим следующую программу. Сначала она создает небольшой вектор, состоящий из целых чисел, а затем использует объект класса **insert_iterator** для вставки новых элементов, не заменяя существующие элементы вектора.

```
// Демонстрация работы класса insert_iterator.
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
```

```
int main()
{
  vector<int> v;
  vector<int>::iterator itr;
  int i;

  for(i=0; i<5; i++)
    v.push_back(i);

  cout << "Исходный массив: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";
  cout << endl;

  itr = v.begin();
  itr += 2; // Ссылка на элемент 2

  // Создаем итератор вставки, ссылающийся на элемент 2.
  insert_iterator<vector<int> > i_itr(v, itr);

  // Происходит вставка, а не замена.
  *i_itr++ = 100;
  *i_itr++ = 200;

  cout << "Массив после вставки: ";
  itr = v.begin();
  while(itr != v.end())
    cout << *itr++ << " ";

  return 0;
}
```

Результаты работы этой программы представлены ниже.

```
Исходный массив: 0 1 2 3 4
Массив после вставки: 0 1 100 200 2 3 4
```

Если ли бы для присвоения элементов 100 и 200 программа использовала обычный итератор, соответствующие элементы в исходном содержании массива были бы перезаписаны. Работа итераторов, описанных классами **back_insert_iterator** и **front_insert_iterator**, основана на тех же принципах.

Класс back_insert_iterator

Класс **back_insert_iterator** поддерживает работу итераторов вывода, вставляющих объекты в конец контейнера с помощью функции **push_back()**. Его шаблонное определение выглядит следующим образом.

```
template <class Cont> class back_insert_iterator:
   public iterator<output_iterator_tag, void, void, void, void>
```

Здесь параметр **Cont** — это тип контейнера, для которого предназначен итератор. Класс **back_insert_iterator** имеет следующий конструктор.

```
explicit back_insert(Cont &cnt);
```

Здесь параметр *cnt* представляет собой контейнер, для которого предназначен итератор. Все вставки выполняются в конце контейнера.

В классе **back_insert_iterator** определены следующие операторы: "=", "*", "++". Указатель на контейнер хранится в защищенной переменной **container**.

Кроме того, в классе определена функция **back_inserter()**, создающая объект класса **back_insert_iterator**. Ее спецификация представлена ниже.

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont &cnt);
```

Класс front_insert_iterator

Класс **front_insert_iterator** поддерживает работу итераторов вывода, вставляющих объекты в начало контейнера с помощью функции **push_front()**. Его шаблонное определение выглядит так.

```
template <class Cont> class front_insert_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Здесь параметр **Cont** — это тип контейнера, для которого предназначен итератор. Класс **front_insert_iterator** имеет следующий конструктор.

```
explicit front_insert_iterator(Cont &cnt);
```

Здесь параметр *cnt* представляет собой контейнер, для которого предназначен итератор. Все вставки выполняются в начале контейнера.

В классе **front_insert_iterator** определены следующие операторы: "=", "*", "++". Указатель на контейнер хранится в защищенной переменной **container**.

Кроме того, в классе определена функция **front_inserter()**, создающая объект класса **front_insert_iterator**. Ее спецификация представлена ниже.

```
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont &cnt);
```

Класс reverse_iterator

Класс **reverse_insert_iterator** поддерживает работу обратных итераторов. Обратный итератор является антиподом обычного итератора. Например, оператор "++" перемещает обратный итератор назад, а не вперед. Его шаблонное определение выглядит так.

```
template <class Cont> class reverse_iterator:
  public iterator<iterator_traits<Iter>::iterator_category,
                  iterator_traits<Iter>::value_type,
                  iterator_traits<Iter>::difference_type,
                  iterator_traits<Iter>::pointer,
                  iterator_traits<Iter>::reference>
```

Здесь параметр **Iter** — это тип итератора произвольного доступа. Класс **reverse_iterator** имеет следующий конструктор.

```
reverse_iterator();
explicit reverse_iterator(Iter itr);
```

Здесь параметр *itr* — это итератор, определяющий стартовую позицию в контейнере.

Если класс **Iter** определяет итератор произвольного доступа, становятся доступными следующие операции: **->**, **+**, **++**, **-**, **--**, *****, **<**, **>**, **<=**, **>=**, **-=**, **+=**, **==**, **!=** и **[]**. Если класс **Iter** определяет двунаправленный итератор, доступными являются лишь операции **->**, **++**, **--**, *****, **==** и **!=**.

В классе **reverse_iterator** определен защищенный член **current**, представляющий собой итератор, ссылающийся на текущую позицию.

Кроме того, в классе **reverse_iterator** определена функция **base()**, спецификация которой имеет следующий вид.

```
Iter base() const;
```

Она возвращает итератор, ссылающийся на текущую позицию.

Класс istream_iterator

Класс **istream_iterator** поддерживает работу потоковых итераторов ввода. Его шаблонное определение выглядит следующим образом.

```
template <class T, class CharType,
          class Attr = char_traits<CharType>,
          class Dist = ptrdiff_t> class istream_iterator:
  public iterator<input_iterator_tag, T, Dist, const T*,
                  const T&>
```

Здесь параметр **T** — это тип передаваемых данных, а параметр **CharType** является символьным типом (**char** или **wchar_t**), которым оперирует поток. Тип **Dist** позволяет хранить разность между двумя адресами. Класс **istream_iterator** имеет следующие конструкторы.

```
istream_iterator();
istream_iterator(istream_type &stream);
istream_iterator(const istream_iterator<T, CharType, Attr, Dist> &ob);
```

Первый конструктор создает итератор, ссылающийся на пустой поток. Второй конструктор создает итератор, ссылающийся на поток *stream*. Тип **istream_type** определяется оператором **typedef** и задает тип потока ввода. Третий конструктор создает копию объекта класса **istream_iterator**.

В классе **istream_iterator** определены следующие операторы: "=", "*", "++". Кроме того, в нем определены операции == и != для объектов класса **istream_iterator**.

Рассмотрим небольшую программу, иллюстрирующую класс **istream_iterator**. Она считывает из потока **cin** и выводит на экран символы, пока не встретится точка.

```
// Применение класса istream_iterator.
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  istream_iterator<char> in_it(cin);

  do {
    cout << *in_it++;
  } while (*in_it != '.');

  return 0;
}
```

Класс istreambuf_iterator

Класс **istreambuf_iterator** поддерживает работу потоковых итераторов ввода символов. Его шаблонное определение выглядит так.

```
template <class CharType, class Attr = char_traits<CharType> >
class istreambuf_iterator:
  public iterator<input_iterator_tag, CharType,
                  typename Attr::off_type, CharType *,
                  CharType &>
```

Здесь параметр **CharType** является символьным типом (**char** или **wchar_t**), которым оперирует поток. Класс **istreambuf_iterator** имеет следующие конструкторы.

```
istreambuf_iterator() throw();
istreambuf_iterator(istream_type &stream) throw();
istreambuf_iterator(streambuf_type *streambuf) throw();
```

Первый конструктор создает итератор, ссылающийся на пустой поток. Второй конструктор создает итератор, ссылающийся на поток *stream*. Тип **istream_type** определяется оператором **typedef** и задаст тип потока ввода. Третий конструктор создает итератор, используя потоковый буфер *streambuf*.

В классе **istreambuf_iterator** определены следующие операторы: "*", "++". Кроме того, в нем определены операции == и != для объектов класса **istreambuf_iterator**.

Класс **istreambuf()** содержит функцию-член **equal()**, спецификация которой имеет следующий вид.

```
bool equal(istreambuf_iterator<CharType, Attr> &ob);
```

Эта операция выглядит довольно противоестественно. Она возвращает значение **true**, если вызывающий итератор и объект *ob* одновременно ссылаются на конец потока. Кроме того, она возвращает значение **true**, если вызывающий итератор и объект *ob* одновременно не ссылаются на конец потока. При этом совершенно не обязательно, чтобы оба итератора ссылались на одну и ту же точку. В противном случае она возвращает значение **false**. Операторы == и != работают аналогично.

Класс ostream_iterator

Класс **ostream_iterator** поддерживает работу потоковых итераторов вывода. Его шаблонное определение выглядит так.

```
template <class T, class CharType,
          class Attr = char_traits<CharType> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Здесь параметр **T** — это тип передаваемых данных, а параметр **CharType** является символьным типом (**char** или **wchar_t**), которым оперирует поток. Класс **ostream_iterator** имеет следующие конструкторы.

```
ostream_iterator(ostream_type &stream);
ostream_iterator(ostream_type &stream, const CharType *delim);
ostream_iterator(const ostream_iterator<T, CharType, Attr> &ob);
```

Первый создает итератор, ссылающийся на поток *stream*. Тип **ostream_type** определяется оператором **typedef** и задает тип потока вывода. Второй конструктор создает итератор, ссылающийся на поток *stream*, используя в качестве разделителя символ

delim, который записывается в поток после каждой операции вывода. Третий вид конструктора создает копию объекта класса **ostream_iterator**.

В классе **ostream_iterator** определены следующие операторы: "=", "*", "++".

Рассмотрим небольшую программу, иллюстрирующую работу класса **ostream_iterator**.

```
// Применение класса ostream_iterator.
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  ostream_iterator<char> out_it(cout);

  *out_it = 'X';
  out_it++;
  *out_it = 'Y';
  out_it++;
  *out_it = ' ';

  char str[] = "Итераторы — мощный инструмент.\n";
  char *p = str;

  while(*p) *out_it++ = *p++;

  ostream_iterator<double> out_double_it(cout);
  *out_double_it = 187.23;
  out_double_it++;
  *out_double_it = -102.7;

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
XY Итераторы — мощный инструмент.
187.23-102.7
```

Класс ostreambuf_iterator

Класс **ostreambuf_iterator** поддерживает работу потоковых итераторов вывода символов. Его шаблонное определение выглядит так.

```
template <class CharType, class Attr = char_traits<CharType> >
class ostreambuf_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Здесь параметр **CharType** является символьным типом (**char** или **wchar_t**), которым оперирует поток. Класс **ostreambuf_iterator** имеет следующие конструкторы.

```
ostreambuf_iterator(ostream_type &stream) throw();
ostreambuf_iterator(streambuf_type *streambuf) throw();
```

Первый конструктор создает итератор, ссылающийся на поток *stream*. Тип **ostream_type** определяется оператором **typedef** и задает тип потока вывода. Второй конструктор создает итератор, используя потоковый буфер *streambuf*. Тип **streambuf_type** определяется оператором **typedef** и задает тип потокового буфера.

В классе **ostreambuf_iterator** определены следующие операторы: "=", "*", "++". Кроме того, класс **ostreambuf()** содержит функцию-член **failed()**, спецификация которой имеет следующий вид.

```
bool failed() const throw();
```

Она возвращает значение **true**, если при выводе никаких ошибок не обнаружено, в противном случае она возвращает значение **true**.

Две функции для работы с итераторами

Существуют две специальные функции, предназначенные для работы с итераторами: **advance()** и **distance()**.

```
template <class InIter, class Dist>
  void advance(InIter &itr, Dist d);
template <class InIter> ptrdiff_t distance(InIter &start, InIter end);
```

Функция **advance()** увеличивает итератор *itr* на величину *d*. Функция **distance()** возвращает количество элементов, расположенных между итераторами *start* и *end*.

Эти две функции необходимы, поскольку только итераторы произвольного доступа позволяют добавлять к итераторам и вычитать из них некую величину. Функции **advance()** и **distance()** позволяют обойти эти ограничения. Однако следует отметить, что некоторые итераторы не обеспечивают эффективную реализацию этих функций.

Функторы

Функторы (function objects) — это классы, в которых определена операторная функция **operator()**. В библиотеке STL определены несколько встроенных функторов. Кроме того, программист может самостоятельно определять свои собственные функторы. Для поддержки функторов необходим заголовок **<functional>**. В нем определены вспомогательные сущности, обеспечивающие работу функторов: редакторы связей (binders), инверторы (negators) и адаптеры (adaptors). Каждая из этих сущностей описывается ниже.

На заметку *Обзор функторов содержится в главе 24.*

Функторы

Функторы разделяются на две категории: бинарные и унарные. Встроенные бинарные функторы перечислены ниже.

plus	**minus**	**multiplies**	**divides**	**modulus**
equal_to	**not_equal_to**	**greater**	**greater_equal**	**less**
less_equal	**logical_and**	**logical_or**		

К унарным встроенным функторам относятся следующие.

logical_not **negate**

Общий вид вызова функтора выглядит так.

```
func_ob<type>()
```

Например, в следующем фрагменте функтор **less()** применяется к операндам типа **int**.

```
less<int>()
```

Базовым для всех бинарных функторов является класс **binary_function**.

```
template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

Базовым для всех унарных функторов является класс **unary_function**.

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};
```

Эти шаблонные классы конкретизируют обобщенные типы данных, используемые функторами. Хотя с технической точки зрения они играют лишь вспомогательную роль, их всегда используют при создании функторов.

Шаблонные спецификации всех бинарных и унарных функторов аналогичны. Рассмотрим их примеры.

```
template <class T> struct plus : binary_function<T, T, T>
{
  T operator() (const T &arg1, const T &arg2) const;
};

template <class T> struct negate : unary_function<T, T>
{
  T operator() (const T &arg) const;
};
```

Каждая функция **operator()** возвращает конкретный результат.

Редакторы связей

Редактор связей обеспечивает связывание значения с аргументом бинарного функтора, создавая унарный функтор. Существуют два редактора связей: **bind1st()** и **bind2nd()**. Они определены следующим образом.

```
template <class BinFunc, class T>
  binder1st<BinFunc> bind1st(const BinFunc &op, const T &value);
template <class BinFunc, class T>
  binder2nd<BinFunc> bind2nd(const BinFunc &op, const T &value);
```

Здесь параметр *op* является бинарным функтором, например **less()** или **greater()**, выполняющим требуемую операцию, а параметр *value* представляет собой связываемое значение. Функтор **bind1st()** возвращает унарный функтор, в котором левый операнд бинарного функтора *op* связан с конкретным значением *value*. Функтор **bind2nd()** возвращает унарный функтор, в котором конкретное значение *value* связывается с правым операндом бинарного функтора *op*. Функтор **bind2nd()** используется намного чаще. В обоих случаях результатом работы редактора связей является унарный функтор, связанный с указанным значением.

Определения классов **binder1st** и **binder2nd** выглядят следующим образом.

```
template <class BinFunc> class binder1st:
  public unary_function(typename BinFunc::second_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::first_argument_type value;
public:
  binder1st(const BinFunc &op,
            const typename BinFunc::first_argument_type &v);
  result_type operator()(const argument_type &v) const;
};

template <class BinFunc> class binder2nd:
  public unary_function(typename BinFunc::first_argument_type,
                        typename BinFunc::result_type>
{
protected:
  BinFunc op;
  typename BinFunc::second_argument_type value;
public:
  binder2nd(const BinFunc &op,
            const typename BinFunc::second_argument_type &v);
  result_type operator()(const argument_type &v) const;
};
```

Здесь класс *BinFunc* является типом бинарного функтора. Обратите внимание на то, что оба класса являются производными от класса **unary_function**. Благодаря этому результирующие объекты функторов **bind1st()** и **bind2nd()** можно применять везде, где допускаются унарные функции.

Инверторы

Инвертор возвращает предикат, значение которого противоположно значению модифицируемого предиката. Существуют два инвертора: **not1()** и **not2()**. Их спецификации выглядят следующим образом.

```
template <class UnPred> unary_negate<UnPred> not1(const UnPred &pred);
template <class BinPred> binary_negate<BinPred> not2(const BinPred &pred);
```

Определения этих классов приведены ниже.

```
template <class UnPred> class unary_negate:
  public unary_function<typename UnPred::argument_type, bool>
{
public:
  explicit unary_negate(const UnPred &pred);
  bool operator()(const argument_type &v) const;
};

template <class UnPred> class binary_negate:
  public binary_function<typename BinPred::first_argument_type,
                         typename BinPred::first_argument_type,
                         bool>
```

```
{
public:
  explicit binary_negate(const BinPred &pred);
  bool operator()(const first_argument_type &v1,
                  const second_argument_type &v2) const;
};
```

В обоих вариантах операторная функция **operator()** возвращает отрицание предиката, заданного параметром *pred*.

Адаптеры

В заголовке **<functional>** определены несколько классов, называемых *адаптерами*. Они позволяют преобразовать указатель на функцию так, чтобы его можно было использовать в стандартной библиотеке шаблонов. Например, с помощью адаптера можно использовать такую функцию, как **strcmp()**, в качестве предиката. Кроме того существуют адаптеры для указателей на члены классов.

Адаптеры указателей на функции

Адаптеры указателей на функцию выглядят так.

```
template <class Argument, class Result>
  pointer_to_unary_function<Argument, Result>
    ptr_fun(Result(*func)(Argument));
template <class Argument1, class Argument2, class Result>
  pointer_to_binary_function<Argument1, Argument2, Result>
    ptr_fun(Result(*func)(Argument1, Argument2));
```

Здесь функция **ptr_fun()** возвращает объект класса **pointer_to_unary_function** или **pointer_to_binary_function**. Эти классы приведены ниже.

```
template <class Argument, class Result>
class pointer_to_unary_function:
  public unary_function<Argument, Result>
{
public:
  explicit pointer_to_unary_function(Result (*func)(Argument));
  Result operator()(Argument arg) const;
};

template <class Argument1, class Argument2, class Result>
class pointer_to_binary_function:
  public binary_function<Argument1, Argument2, Result>
{
public:
  explicit pointer_to_binary_function(
             Result (*func)(Argument1, Argument2));
  Result operator()(Argument1 arg1, Argument2 arg2) const;
};
```

Для унарных функций операторная функция **operator()** возвращает значение *func*(*arg*), а для бинарных — значение *func*(*arg1*, *arg2*).

Типы результатов этих операций задаются обобщенным типом **Result**.

Адаптеры указателей на функции — члены класса

Адаптеры указателей на функцию — член класса выглядят так.

```
template <class Result, class T>
  mem_fun_t<Result, T> mem_fun(Result(T::*func)());
template <class Result, class Argument>
  mem_fun1_t<Result, T, Argument>
    mem_fun1(Result(T::*func)(Argument));
```

Здесь функция **mem_fun()** возвращает объект класса **mem_fun_t**, а функция **mem_fun1()** возвращает объект класса **mem_fun1_t**. Эти классы приведены ниже.

```
template <class Result, class T> class mem_fun_t:
  public unary_function<T*, Result> {
public:
  explicit mem_fun_t(Result (T::*func)());
  Result operator()(T *func) const;
};

template <class Result, class T,
          class Argument> class mem_fun1_t:
  public binary_function<T*, Argument, Result> {
public:
  explicit mem_fun1_t(Result (T::*func)(Argument));
  Result operator()(T *func, Argument arg) const;
};
```

Конструктор класса **mem_fun_t** вызывает функцию-член, заданную в качестве параметра. Конструктор класса **mem_fun1_t** вызывает функцию-член, заданную в качестве первого параметра, передавая ей как аргумент свой второй параметр, имеющий тип **Argument**.

Для ссылок на функции-члены существует параллельный набор адаптеров.

```
template <class Result, class T>
  mem_fun_ref_t<Result, T> mem_fun_ref(Result(T::*func)());
template <class Result, class T, class Argument>
  mem_fun1_ref_t<Result, T, Argument>
    mem_fun1_ref(Result(T::*func)(Argument));
```

Эти классы приведены ниже.

```
template <class Result, class T> class mem_fun_ref_t:
  public unary_function<T, Result> {
public:
  explicit mem_fun_ref_t(Result (T::*func)());
  Result operator()(T &func) const;
};

template <class Result, class T,
          class Argument> class mem_fun1_ref_t:
  public binary_function<T, Argument, Result> {
public:
  explicit mem_fun1_ref_t(Result (T::*func)(Argument));
  Result operator()(T &func, Argument arg) const;
};
```

Распределители памяти

Распределитель памяти управляет выделением памяти для контейнера. Поскольку в библиотеке STL определен распределитель памяти по умолчанию, который автоматически используется контейнерами, большинство программистов избавлено от необходимости разбираться в механизме его работы или создавать свой собственный распределитель. Однако такие значения оказываются полезными, если необходимо создать свою библиотеку классов или нечто подобное.

Все распределители памяти должны удовлетворять нескольким требованиям. Во-первых, они должны содержать определения следующих типов.

const_pointer	Константный указатель на объект класса **value_type**.
const_reference	Константная ссылка на объект класса **value_type**.
difference_type	Разность между двумя адресами.
pointer	Указатель на объект класса **value_type**.
reference	Ссылка на объект класса **value_type**.
size_type	Тип, позволяющий хранить размер наибольшего возможного объекта, размещаемого в памяти.
value_type	Тип объекта, размещаемого в памяти.

Во-вторых, все распределители памяти должны предусматривать следующие функции.

address	Возвращает адрес по заданной ссылке.
allocate	Выделяет память.
deallocate	Освобождает память.
max_size	Возвращает максимальное количество объектов, которые можно разместить в памяти.
construct	Создает объект.
destroy	Разрушает объект.

Кроме того, в этих классах должны быть определены операторы "==" и "!=".

Распределитель памяти, предусмотренный по умолчанию, называется **allocator**. Этот класс определен в заголовке **<memory>**. Его шаблонная спецификация выглядит следующим образом.

```
template <class T> class allocator
```

Здесь класс **T** представляет собой тип объектов, размещаемых в памяти с помощью распределителя **allocator**. В классе **allocator** определяются следующие конструкторы.

```
allocator() throw();
allocator(const allocator<T> &ob) throw();
```

Первый конструктор создает новый распределитель памяти, второй — копию объекта *ob*.

Кроме того, в классе **allocator** определены операторы "==" и "!=". Функции — члены класса **allocator** перечислены в табл. 35.2.

Таблица 35.2. Функции — члены класса allocator

Функция	*Описание*
pointer address(reference *ob***) const;** **const_pointer address(const_reference** *ob***) const;**	Возвращает адрес объекта *ob*

Окончание табл. 35.2

Функция	Описание
`pointer allocate(size_type num, allocator<void>::const_pointer h = 0);`	Возвращает указатель на выделенную память, размер которой достаточен для хранения *num* объектов типа `T`. Значение параметра *h* подсказывает функции размер, выделяемый по умолчанию, а также позволяет задать произвольный размер выделяемой памяти
`void construct(pointer ptr, const_reference val);`	Создает объект типа `T`, инициализируя его значением, заданным ссылкой *val*
`void deallocate(pointer ptr, size_type num);`	Удаляет из памяти *num* объектов типа `T`, начиная с указателя *ptr*. Значение указателя *ptr* необходимо получить с помощью функции `allocate()`
`void destroy(pointer ptr);`	Уничтожает объект, на который ссылается указатель *ptr*. Его деструктор вызывается автоматически
`size_type max_size() const throw();`	Возвращает максимальное количество объектов типа `T`, которое можно разместить в памяти

И последнее замечание: существует также специализация класса `allocator` для указателей типа `void*`.

Полный справочник по

Глава 36

Класс string

В этой главе описывается стандартный класс **string**. Язык C++ поддерживает два способа работы со строками. Первый способ основан на применении массивов символов, заканчивающихся нулевым байтом. Иногда такой массив называют *C-строкой*. Второй способ использует шаблонный класс **basic_string**. Этот класс имеет две специализации: класс **string**, обеспечивающий работу со строками, состоящими из символов типа **char**, и класс **wstring**, поддерживающий работу со строками расширенных символов, имеющих тип **wchar_t**.Чаще всего используется класс **string**.

Класс **basic_string**, по существу, представляет собой контейнер. Это значит, что к строкам можно применять итераторы и стандартные алгоритмы. Однако строки обладают дополнительными возможностями.

Класс **basic_string** использует класс **char_traits**, в котором определены некоторые атрибуты символов, образующих строку. Несмотря на то что большинство строк состоят из символов типа **char** или **wchar_t**, класс **basic_string** может оперировать любыми объектами, которые можно использовать для представления текстовых символов. Классы **basic_string** и **char_traits** описаны ниже.

На заметку *Обзор класса* ***string*** *приведен в главе 24.*

Класс basic_string

Шаблонная спецификация класса **basic_string** имеет следующий вид.

```
template <class CharType, class Attr = char_traits<CharType>,
          class Allocator = allocator<T> > class basic_string
```

Здесь класс **CharType** определяет тип используемых символов. Класс **Attr** описывает атрибуты символов, а класс **Allocator** задает распределитель памяти. Класс **basic_string** имеет следующие конструкторы.

```
explicit basic_string(const Allocator &a = Allocator());
basic_string(size_type len, CharType ch,
             const Allocator &a = Allocator());
basic_string(const CharType *str, const Allocator &a = Allocator());
basic_string(const CharType *str, size_type len,
             const Allocator &a = Allocator());
basic_string(const basic_string &str, size_type indx = 0,
             size_type len = npos, const Allocator &a = Allocator());
template <class InIter> basic_string(InIter start, InIter end,
             const Allocator &a = Allocator());
```

Первый конструктор создает пустую строку, второй — строку, содержащую *len* символов, имеющих значение *ch*. Третий конструктор создает объект, являющийся копией строки *str*. Четвертый генерирует подстроку строки *str*, начинающуюся с нуля и состоящую из *len* символов. Пятый конструктор создает строку из другого объекта класса **basic_string**, используя подстроку, начинающуюся с индекса *indx* и состоящую из *len* символов. Шестой генерирует строку, содержащую элементы диапазона, ограниченного итераторами *start* и *end*.

В классе **basic_string** определены следующие операции:

```
==, <, <=, !=, >, >+
```

Кроме того, в нем определены оператор "+", результатом которого является конкатенация двух строк, и операторы вывода "<<" и ">>".

Оператор "+" позволяет конкатенировать два объекта класса **string**, а также объект класса **string** и С-строку. Иначе говоря, поддерживаются следующие варианты.

string+string
strinf+С-строка
С-строка+string

Кроме того, с помощью оператора "+" к концу строки можно приписать еще один символ.

В классе **basic_string** определена константа **npos**, равная -1. Эта константа задает максимально возможную длину строки.

Обобщенный класс **CharType** описывает тип символов, хранящихся в строке. Поскольку имена символов-заполнителей в шаблонном классе произвольны, класс **basic_string** переопределяет их с помощью оператора **typedef**. В классе **basic_string** определены следующие типы.

size_type	Разновидность целочисленного типа, отдаленно напоминающая тип **size_t**.
reference	Ссылка на символ, содержащийся в строке.
const_reference	Константная ссылка на символ, содержащийся в строке.
iterator	Итератор.
const_iterator	Константный итератор.
reverse_iterator	Обратный итератор.
const_reverse_iterator	Константный обратный итератор.
value_type	Тип символа, содержащегося в строке.
allocator_type	Тип распределителя памяти.
pointer	Указатель на символ, содержащийся е строке.
const_pointer	Константный указатель на символ, содержащийся в строке.
traits_type	Переопределение типа **char_traits<CharType>**.
difference_type	Тип, предназначенный для хранения разности двух адресов.

Функции-члены, определенные в классе **basic_string**, перечислены в табл. 36.1. Поскольку подавляющее большинство программистов используют обычные символьные строки (а также для простоты изложения), таблица ссылается лишь на класс **string**, хотя эти функции можно применять и к объектам класса **wchar_t** (или любого другого класса, который позволяет представлять текстовые символы)

Таблица 36.1. Функции — члены класса string

string &append(const string &*str***);**	Добавляет строку *str* в конец еызывающей строки. Возвращает указатель ***this**
string &append(const string &*str* **size_type** *indx*, **size_type** *len*);	Добавляет подстроку строки *str* в конец вызывающей строки. Добавляемая подстрока начинается с индекса *indx* и состоит из *len* символов. Возвращает указатель ***this**
string &append(const CharType &*str***);**	Добавляет строку *str* в конец вызывающей строки. Возвращает указатель ***this**
string &append(const CharType &*str*, **size_type** *num*);	Добавляет *num* символов строки *str* в конец вызывающей строки. Возвращает указатель ***this**
string &append(size_type *len*, **CharType** *ch*);	Добавляет *len* символов, заданных параметром *ch*, в конец вызывающей строки. Возвращает указатель ***this**

`template<class InIter> string &append(InIter start, InIter end);`	Добавляет последовательность символов, ограниченную итераторами *start* и *end*, в конец вызывающей строки. Возвращает указатель `*this`
`string &assign(const string &str);`	Присваивает строку *str* вызывающей строке. Возвращает указатель `*this`
`string &assign(const string &str, size_type indx, size_type len);`	Присваивает вызывающей строке подстроку строки *str*. Присеаиеаемая подстрока начинается с индекса *indx* и состоит из *len* символов. Возвращает указатель `*this`
`string &assign(const CharType &str);`	Присваиеает строку *str* вызывающей строке. Возвращает указатель `*this`
`string &assign(const CharType &str, size_type len);`	Присваивает вызывающей строке первые *len* символов строки *str*. Возвращает указатель `*this`
`string &assign(size_type len, CharType ch);`	Приписывает *len* символов, заданных параметром *ch*, е конец вызыеающей строки. Возвращает указатель `*this`
`template<class InIter> string &assign(InIter start, InIter end);`	Присваивает вызывающей строкепоследовательность символов, ограниченную итераторами *start* и *end*. Возвращает указатель `*this`
`reference at(size_type indx);` `const_reference at(size_type indx) const;`	Возвращает ссылку на символ, заданный параметром *indx*
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор, установленный на первый элемент строки
`const CharType *c_str() const;`	Возвращает указатель на С-строку (т.е. массив символов, заеершающийся нулевым байтом), представляющий собой версию вызывающего объекта
`size_type capacity() const;`	Возвращает текущий размер строки, равный количеству символов, которое она способна содержать без дополнительного выделения памяти
`int compare(const string &str) const;`	Сравнивает строку *str* с вызывающей строкой. Возвращаются следующие значения: меньше нуля, если `*this`<*str*, нуль, если `*this`=*str*, больше нуля, если `*this`>*str*
`int compare(size_type indx, size_type len, const string &str) const;`	Сравнивает строку *str* с подстрокой вызывающей строки. Подстрока начинается с индекса *indx* и состоит из *len* символов. Возвращаются следующие значения: меньше нуля, если `*this`<*str*, нуль, если `*this`=*str*, больше нуля, если `*this`>*str*
`int compare(size_type indx, size_type len, const string &str, size_type indx2, size_type len2) const;`	Сравнивает подстроку строки *str* с подстрокой вызывающей строки. Подстрока начинается с индекса *indx* и состоит из *len* символов. Подстрока строки *str* начинается с индекса *indx2* и состоит из *len2* символов. Возвращаются следующие значения: меньше нуля, если `*this`<*str*, нуль, если `*this`=*str*, больше нуля, если `*this`>*str*

Продолжение табл. 36.1

`int compare(const CharType &str) const;`	Сравнивает строку *str* с вызывающей строкой. Возвращаются следующие значения: меньше нуля, если `*this<str`, нуль, если `*this=str`, больше нуля, если `*this>str`

`int compare(size_type indx,`
`size_type len,`
`const CharType &str,`
`size_type len2= npos) const;`
	Сравнивает подстроку строки *str* с подстрокой вызывающей строки. Подстрока начинается с индекса *indx* и состоит из *len* символов. Подстрока строки *str* начинается с нуля и состоит из *len2* символов. Возвращаются следующие значения:
	меньше нуля, если `*this<str`,
	нуль, если `*this=str`,
	больше нуля, если `*this>str`

`size_type copy(CharType *str,`
`size_type len,`
`size_type indx= 0) const;`
	Копирует *len* символов из вызывающей строки в символьный массив *str*, начиная с индекса *indx*

`const CharType *data() const;`	Возвращает указатель на первый символ вызывающей строки
`bool empty() const;`	Возвращает значение `true`, если вызывающая строка пуста, и `false` — в противном случае
`iterator end();` `const_iterator end() const;`	Возвращает итератор, установленный на конец строки
`iterator erase(iterator i);`	Удаляет символ, на который ссылается итератор *i*. Возвращает итератор на следующий символ
`iterator erase(iterator start,` `iterator end);`	Удаляет символы из диапазона, ограниченного итераторами *start* и *end*. Возвращает итератор на символ, следующий за последним удаленным символом
`string &erase(size_type indx = 0,` `size_type len = npos);`	Удаляет *len* символов из вызывающей строки, начиная с индекса *indx*. Возвращает указатель `*this`
`size_type find(const string &str,` `size_type indx = 0) const;`	Возвращает индекс первого ехождения строки *str* в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find(const CharType &str,` `size_type indx = 0) const;`	Возвращает индекс первого вхождения строки *str* в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`

`size_type find(const CharType &str,`
`size_type indx,`
`size_type len ) const;`
	Возвращает индекс первого вхождения первых *len* символов строки *str* в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`

`size_type find(CharType ch,` `size_type indx= 0) const;`	Возвращает индекс переого вхождения символа в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_first_of(const string &str, size_type indx = 0) const;`	Возвращает индекс первого символа вызывающей строки, совпадающего с каким-либо символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_ type find_first_of(const CharType *str, size_type indx = 0) const`	Возвращает индекс первого символа вызывающей строки, соепадающего с каким-либо символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`

Продолжение табл. 36.1

`size_type find_first_of(const CharType *str, size_type indx, size_type len) const;`	Возвращает индекс первого символа вызывающей строки, совпадающего с каким-либо символом из первых *len* символов строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_first_of(CharType ch, size_type indx = 0) const;`	Возвращает индекс первого вхождения символа *ch* в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_first_not_of (const string &str, size_type indx = 0) const;`	Возвращает индекс первого символа вызывающей строки, не совпадающего ни с одним символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, еозвращается значение `npos`
`size_type find_first_not_of (const CharType *str, size_type indx = 0) const;`	Возвращает индекс первого символа вызывающей строки, не совпадающего ни с одним символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_first_not_of (const CharType *str, size_type indx, size_type len) const;`	Возвращает индекс первого символа вызывающей строки, не совпадающего ни с одним из первых *len* символов строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_first_not_of ( CharType ch, size_type indx=0) const;`	Возвращает индекс первого символа вызывающей строки, не совпадающего с символом *ch*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_last_of(const string &str, size_type indx = npos) const;`	Возвращает индекс последнего символа вызывающей строки, совпадающего с каким-либо символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_last_of (const CharType *str, size_type indx = npos) const;`	Возвращает индекс последнего символа вызывающей строки, совпадающего с каким-либо символом из первых *len* символое строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_last_of (const CharType *str, size_type indx, size_type len) const;`	Возвращает индекс последнего символа вызывающей строки, совпадающего с каким-либо символом из первых *len* символов строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_last_of(CharType ch, size_type indx = npos) const;`	Возвращает индекс последнего вхождения символа *ch* в вызывающую строку. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`size_type find_last_not_of (const string &str, size_type indx = npos) const;`	Возвращает индекс последнего символа вызывающей строки, не совпадающего ни с одним символом строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, еозвращается значение `npos`
`size_type find_last_not_of (const CharType *str, size_type indx = npos) const;`	Возвращает индекс последнего символа вызывающей строки, не совпадающего ни с одним из символов строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`

`size_type find_last_not_of(`
`  const CharType *str,`
`  size_type indx,`
`  size_type len) const;`
	Возвращает индекс последнего символа вызывающей строки, не совпадающего ни с одним из первых *len* символов строки *str*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`

`size_type find_last_not_of(CharType ch,` `size_type indx = npos) const;`	Возвращает индекс последнего символа вызывающей строки, не совпадающего с символом *ch*. Поиск начинается с позиции *indx*. Если совпадение не обнаружено, возвращается значение `npos`
`allocator_type get_allocator() const;`	Возвращает распределитель памяти для строки
`iterator insert(iterator i,` `  const CharType &ch);`	Вставляет символ *ch* непосредственно перед символом, заданным итератором *i*. Возвращает итератор, установленный на этот символ
`string &insert(size_type indx,` `  const string &str);`	Вставляет строку *str* в вызывающую строку, начиная с индекса *indx*. Возвращает указатель `*this`

`string &insert(size_type indx1,`
`  const string &str,`
`  size_type indx2,`
`  size_type len);`
	Вставляет подстроку строки *str* в вызывающую строку, начиная с индекса *indx1*. Подстрока начинается с индекса *indx2* и состоит из *len* символов. Возвращает указатель `*this`

`string &insert(size_type indx,` `  const CharType *str);`	Вставляет строку *str* в вызывающую строку, начиная с индекса *indx*. Возвращает указатель `*this`

`string &insert(size_type indx,`
`  const CharType *str,`
`  size_type len);`
	Вставляет первые *len* символов строки *str* в вызывающую строку, начиная с индекса *indx*. Возвращает указатель `*this`

`string &insert(size_type indx,`
`  size_type len,`
`  CharType ch);`
	Вставляет в вызывающую строку первые *len* символов, заданных параметром *ch*, начиная с индекса *indx*. Возвращает указатель `*this`

`void insert(iterator i,`
`  size_type len,`
`  const CharType &ch);`
	Вставляет в вызывающую строку *len* копий символа, заданного параметром *ch*, перед элементом, указанным итератором *i*

`template <class InIter>`
`  void insert(iterator i, InIter start,`
`  InIter end);`
	Вставляет последовательность, ограниченную итераторами *start* и *end*, непосредственно перед элементом, на который ссылается итератор *i*

`size_type length() const;`	Возвращает количество символов, содержащихся в строке
`size_type max_size() const;`	Возвращает максимальное количество символов, которые могут содержаться в строке

`reference operator[](size_type indx) const;`
`const_reference operator[]`
`  (size_type indx) const;`
	Возвращает ссылку на символ с индексом *indx*

`string &operator=(const string &str);`
`string &operator=(const CharType &str);`
`string &operator=(const CharType ch);`
	Присваивает указанную строку или символ вызывающей строке. Возвращает указатель `*this`

`string &operator+=(const string &str);`
`string &operator+=(const CharType *str);`
`string &operator+=( CharType ch);`
	Добавляет указанную строку или символ в конец вызывающей строки. Возвращает указатель `*this`

`void push_back(const CharType ch);`	Добавляет символ *ch* в конец вызывающей строки
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор, установленный на конец строки

`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор, установленный на начало строки

`string &replace(size_type` *indx*`,`
`size_type` *len*`,`
`const string &`*str*`);`
	Заменяет до *len* символов вызывающей строки символами строки *str*, начиная с индекса *indx*. Возвращает указатель `*this`

`string &replace(size_type` *indx1*`,`
`size_type` *len1*`,`
`const string &`*str*`,`
`size_type` *indx2*`,`
`size_type` *len2*`);`
	Заменяет до *len1* символов вызывающей строки, начиная с индекса *indx1*, *len2* символами строки *str*, начиная с индекса *indx2*. Возвращает указатель `*this`

`string &replace(size_type` *indx*`,`
`size_type` *len*`,`
`const CharType *`*str*`);`
	Заменяет до *len* символов вызывающей строки символами строки *str*, начиная с индекса *indx*. Возвращает указатель `*this`

`string &replace(size_type` *indx1*`,`
`size_type` *len1*`,`
`const CharType *`*str*`,`
`size_type` *len2*`);`
	Заменяет до *len1* символов вызывающей строки, начиная с индекса *indx1*, *len2* символами строки *str*. Возвращает указатель `*this`

`string &replace(size_type` *indx*`,`
`size_type` *len1*`,`
`size_type` *len2*`,`
`CharType` *ch*`);`
	Заменяет до *len1* символов вызывающей строки, начиная с индекса *indx1*, *len2* копиями символа *ch*. Возвращает указатель `*this`

`string &replace(iterator` *start*`,`
`iterator` *end*`,`
`const string &`*str*`);`
	Заменяет диапазон символов, ограниченный итераторами *start* и *end*, строкой *str*. Возвращает указатель `*this`

`string &replace(iterator` *start*`,`
`iterator` *end*`,`
`const CharType *`*str*`);`
	Заменяет диапазон символов, ограниченный итераторами *start* и *end*, строкой *str*. Возвращает указатель `*this`

`string &replace(iterator` *start*`,`
`iterator` *end*`,`
`const CharType *`*str*`,`
`size_type` *len*`);`
	Заменяет диапазон символов, ограниченный итераторами *start* и *end*, первыми *len* символами строки *str*. Возвращает указатель `*this`

`string &replace(iterator` *start*`,`
`iterator` *end*`,`
`size_type` *len*`,`
`CharType` *ch*`);`
	Заменяет диапазон символов, ограниченный итераторами *start* и *end*, *len* копиями символа *ch*. Возвращает указатель `*this`

`template <class InIter>`
`string &replace(iterator` *start*`,`
`iterator` *end1*`,`
`iterator` *start2*`,`
`iterator` *end2*`);`
	Заменяет диапазон символов, ограниченный итераторами *start1* и *end1*, диапазоном символов, ограниченным итераторами *start2* и *end2*. Возвращает указатель `*this`

`void reserve(size_type` *num* `= 0);`	Задает размер строки так, чтобы он был не меньше значения *num*
`void resize(size_type` *num*`);` `void resize(size_type` *num*`, CharType` *ch*`);`	Меняет размер строки на значение *num*. Если строку необходимо удлинить, в ее конец добавляются символы *ch*
`size_type rfind(const string &`*str*`,` `size_type` *indx* `= npos) const;`	Возвращает индекс последнего вхождения строки *str* в вызывающую строку. Поиск заканчивается на индексе *indx*. Если совпадение не найдено, возвращается значение `npos`
`size_type rfind(const CharType *`*str*`,` `size_type` *indx* `= npos) const;`	Возвращает индекс последнего вхождения строки *str* в вызывающую строку. Поиск заканчивается на индексе *indx*. Если совпадение не найдено, возвращается значение `npos`

Окончание табл. 36.1

`size_type rfind(const CharType *str, size_type indx, size_type len) const;`	Возвращает индекс последнего вхождения первых *len* символов строки *str* в вызывающую строку. Поиск заканчивается на индексе *indx*. Если совпадение не найдено, возвращается значение `npos`
`size_type rfind(CharType ch, size_type indx = npos) const;`	Возвращает индекс последнего вхождения символа *ch* в вызывающую строку. Поиск заканчивается на индексе *indx*. Если совпадение не найдено, возвращается значение `npos`
`size_type size() const;`	Возвращает текущее количество символов в строке
`string substr(size_type indx = 0, size_type len = npos) const;`	Возвращает подстроку вызывающей строки, состоящую из *len* символов, начиная с индекса *indx*
`void swap(string &str);`	Меняет местами символы вызывающей строки и символы, хранящиеся в объекте *str*

Класс char_traits

Класс `char_traits` описывает атрибуты символов. Его шаблонная спецификация выглядит следующим образом.

```
template <class CharType> struct char_traits
```

Здесь класс `CharType` описывает тип символов.

Библиотека языка C++ содержит две специализации класса `char_traits`: для обычных символов (тип `char`) и для расширенных символов (тип `wchar_t`). В классе `char_traits` определены пять типов данных.

`char_type`	Тип символа, синоним класса `CharType`.
`int_type`	Целочисленный тип, способный хранить символ типа `char_type` или признак конца файла.
`off_type`	Целочисленный тип, способный представлять смещение внутри потока.
`pos_type`	Целочисленный тип, способный представлять позицию внутри потока.
`state_type`	Тип объекта, способного хранить состояние преобразования. (Применяется к многобайтовым символам.)

Таблица 36.2. Функции — члены класса char_traits

`static void assign(char_type &ch1, const char_type &ch2);`	Присваивает символ *ch2* символу *ch1*
`static char_type *assign(char_type *str, size_t num, char_type ch2);`	Присваивает символ *ch2* первым *num* символам строки *str*. Возвращает строку *str*
`static int compare(const char_type *str1, const char_type *str2 size_t num);`	Сравнивает *num* символов строки *str1* с соответствующими символами строки *str2*. Если строки совпадают, возвращает нуль. В противном случае возвращается отрицательное число, если *str1*<*str2*, и положительное число, если *str1*>*str2*
`static char_type *copy(char_type *to, const char_type *from, size_t num);`	Копирует *num* символов из строки *from* в строку *to*

`static int_type eof();`	Возвращает признак конца файла
`static bool eq(const char_type &`*ch1*`,` `const char_type &`*ch2*`);`	Сравнивает символы *ch1* и *ch2*. Возвращает значение **true**, если символы совпадают, и значение **false** — если нет

`static bool eq_int_type(`
`const int_type &`*ch1*`,`
`const int_type &`*ch2*`);`
	Возвращает значение **true**, если символы *ch1* и *ch2* эквивалентны, и значение **false** — если нет

`static const char_type *find(`
`const char_type *`*str*`,`
`size_t` *num*`,`
`const char_type *`*ch*`);`
	Возвращает указатель на первое вхождение символа *ch* в строку *str*. Проверяются только первые *num* символов. В случае отрицательного ответа возвращается нулевой указатель

`static size_t length(const chat_type *`*str*`);`	Возвращает длину строки *str*
`static bool lt(const char_type &`*ch1*`,` `const char_type &`*ch2*`);`	Возвращает значение **true**, если *ch1*<*ch2*, и значение **false** в противном случае

`static char_type *move(char_type *`*to*`,`
`const char_type *`*from*`,`
`size_t` *num*`);`
	Копирует *num* символов из строки *from* в строку *to*. Возвращает указатель *to*

`static int_type not_eof(const int_type &`*ch*`);`	Возвращает символ *ch*, если он не равен константе **EOF**. В противном случае возвращается признак конца файла
`static char_type to_char_type` `(const int_type &`*ch*`);`	Преобразовывает параметр *ch* в объект класса **char_type** и возвращает результат
`static int_type to_int_type` `(const char_type &`*ch*`);`	Преобразовывает символ *ch* в объект класса **int_type** и возвращает результат

Полный справочник по

Глава 37

Числовые классы

При стандартизации языка C++ в его состав была включена библиотека числовых классов. Эти классы позволяют разрабатывать вычислительные программы. Некоторые из функций — членов этих классов дублируют функции, унаследованные от библиотеки языка C. Различие между ними заключается в том, что числовые функции, описанные в этой главе, оперируют объектами типа **valarray**, представляющими собой массив чисел, либо объектами класса **complex**, описывающими комплексные числа. Включение числовых классов позволило расширить круг задач, для решения которых можно применять стандартную библиотек языка C++.

Класс complex

Заголовок **<complex>** содержит определение класса **complex**, представляющего комплексные числа. Кроме того, в нем описан набор функций и операторов, работающих с объектами типа **complex**.

Шаблонная спецификация класса **complex** выглядит следующим образом.

```
template <class T> class complex
```

Здесь класс **T** задает тип, используемый для хранения компонентов комплексных чисел. Существуют три специализации класса **complex**.

```
class complex<float>
class complex<double>
class complex<long dounle>
```

Класс **complex** содержит следующие конструкторы.

```
complex(const T &real = T(), const T &imaginary = T());
complex(class complex &ob);
template <class T1> complex(const complex<T1> &ob);
```

Первый конструктор создает объект класса **complex** с действительной частью *real* и мнимой частью *imaginary*. Если не указано иное, по умолчанию эти значения равны нулю. Второй конструктор создает копию объекта *ob*, а третий — объект класса **complex** на основе объекта *ob*.

В классе **complex** определены следующие операции.

+	-	*	/
-=	+=	/=	*=
=	==	!=	

Операторы, не являющиеся операторами присваивания, перегружаются тремя способами. Во-первых, как операции, левым операндом которых является объект класса **complex**, а правым — скалярный объект. Во-вторых, как операции, левым операндом которых является скалярный объект, а правым — объект класса **complex**. В-третьих, как операции, обоими операндами которых являются объекты класса **complex**. Например, допускаются следующие виды операций:

```
объект класса complex + скаляр
скаляр + объект класса complex
объект класса complex + объект класса complex
```

Операции над скалярными объектами влияют лишь на действительную часть числа.

В классе **complex** определены две функции-члена: **real()** и **imag()**, имеющие следующие спецификации.

```
T real() const;
T imag() const;
```

Функция **real()** возвращает действительную часть вызывающего объекта, а функция **imag()** — мнимую часть числа. Функции — члены класса **complex** перечислены в табл. 37.1.

Таблица 37.1. Функции — члены класса complex

`template <class T> T abs(const complex<T> &ob)`	Возвращает абсолютное значение числа *ob*
`template <class T> T arg(const complex<T> &ob)`	Возвращает фазовый угол числа *ob*
`template <class T> complex<T> conj(const complex<T> &ob)`	Возвращает число, сопряженное числу *ob*
`template <class T> complex<T> cos(const complex<T> &ob)`	Возвращает косинус числа *ob*
`template <class T> complex<T> cosh(const complex<T> &ob)`	Возвращает гиперболический косинус числа *ob*
`template <class T> complex<T> exp(const complex<T> &ob)`	Возвращает значение e^{ob}
`template <class T> T imag(const complex<T> &ob)`	Возвращает мнимую часть числа *ob*
`template <class T> complex<T> log(const complex<T> &ob)`	Возвращает натуральный логарифм числа *ob*
`template <class T> complex<T> log10(const complex<T> &ob)`	Возвращает десятичный логарифм числа *ob*
`template <class T> T norm(const complex<T> &ob)`	Возвращает модуль числа *ob*
`template <class T> complex<T> polar(const T &v, const T &theta=0);`	Возвращает комплексное число, модуль которого равен параметру *v*, а фазовый угол — параметру *theta*
`template <class T> complex<T> pow(const complex<T> &b, int e);`	Возвращает значение b^e
`template <class T> complex<T> pow(const complex<T> &b, const T &e);`	Возвращает значение b^e
`template <class T> complex<T> pow(const complex<T> &b, const complex<T> &e);`	Возвращает значение b^e
`template <class T> complex<T> pow(const T &b, const complex<T> &e);`	Возвращает значение b^e
`template <class T> T real(const complex<T> &ob);`	Возвращает действительную часть числа *ob*

Окончание табл. 37.1

`template <class T> complex<T> sin(const complex<T> &ob)`	Возвращает синус числа *ob*
`template <class T> complex<T> sinh(const complex<T> &ob)`	Возвращает гиперболический синус числа *ob*
`template <class T> complex<T> sqrt(const complex<T> &ob)`	Возвращает квадратный корень числа *ob*
`template <class T> complex<T> tan(const complex<T> &ob)`	Возвращает тангенс числа *ob*
`template <class T> complex<T> tanh(const complex<T> &ob)`	Возвращает гиперболический тангенс числа *ob*

Ниже приведена программа, иллюстрирующая применение класса **complex**.

```
// Демонстрация применения класса complex.
#include <iostream>
#include <complex>
using namespace std;

int main()
{
  complex<double> cmpx1(1, 0);
  complex<double> cmpx2(1, 1);

  cout << cmpx1 << " " << cmpx2 << endl;

  complex<double> cmpx3 = cmpx1 + cmpx2;
  cout << cmpx3 << endl;

  cmpx3 += 10;
  cout << cmpx3 << endl;

  return 0;
}
```

Результаты работы этой программы таковы.

```
(1, 0) (1, 1)
(2, 1)
(12, 1)
```

Класс vallaray

Заголовок **<vallaray>** содержит определение большого количества классов, обеспечивающих работу с числовыми массивами. В этом заголовке описано много функций-членов и обычных функций. Класс **vallaray** позволяет решать разнообразные задачи, однако особенно он полезен для программирования вычислений. Кроме того, хотя класс **vallaray** очень велик, большинство его операций интуитивно понятны.

Шаблонная спецификация класса **vallaray** выглядит следующим образом.

```
template <class T> class vallaray
```

Класс **vallaray** содержит следующие конструкторы.

```
vallaray();
explicit vallaray(size_t num);
vallaray(const T &v, size_t num);
vallaray(const T *ptr, size_t num);
vallaray(const vallaray<T> &ob);
vallaray(const slice_array<T> &ob);
vallaray(const gslice_array<T> &ob);
vallaray(const mask_array<T> &ob);
vallaray(const indirect_array<T> &ob);
```

Первый конструктор создаст пустой объект, а второй — объект класса **vallaray**, имеющий длину *num*. Третий конструктор генерирует объект класса **vallaray** длины *num*, инициализированный объектом *v*. Четвертый конструктор создаст объект класса **vallaray** длины *num*, инициализированный объектом, на который ссылается указатель *ptr*. Пятый конструктор генерирует копию объекта *ob*. Остальные четыре конструктора создают объект класса **vallaray** на основе одного из вспомогательных классов.

В классе **vallaray** определены следующие операции.

+	-	*	/
-=	+=	/=	*=
=	==	!=	<<
>>	<<=	>>=	^
^=	%	%=	-
!	|	|=	&
&=	()		

Операторы, не являющиеся операторами присваивания, перегружаются несколькими способами. Функции — члены класса **vallaray** перечислены в табл. 37.2. Обычные функции, работающие с классом **vallaray**, приведены в табл. 37.3. Трансцендентные функции, определенные для класса **vallaray**, указаны в табл. 37.4.

Таблица 37.2. Функции — члены класса vallaray

`vallaray<T> apply(T func(T)) const;` `vallaray<T> apply(T func(const T &ob)) const;`	Применяет функцию *func()* к вызывающему массиву. Возвращает массив, содержащий результат
`vallaray<T> cshift(int num) const;`	Выполняет левое вращение вызывающего массива на *num* позиций (т.е. выполняет циклический сдвиг влево). Возвращает массив, содержащий результат
`T max() const;`	Возвращает максимальное значение, хранящееся в вызывающем массиве
`T min() const;`	Возвращает минимальное значение, хранящееся в вызывающем массиве
`vallaray<T>` `&operator=(const vallaray<T> &ob);`	Присваивает элементы объекта *ob* соответствующим элементам вызывающего массива. Возвращает ссылку на вызывающий массив
`vallaray<T>` `&operator=(const T &v);`	Присваивает каждому элементу вызывающего массива значение *v*. Возвращает ссылку на вызывающий массив
`vallaray<T>` `&operator=(const slice_array<T> &ob);`	Присваивает подмножество. Возвращает ссылку на вызывающий массив

`vallaray<T> &operator=(const gslice_array<T> &ob);`	Присваивает подмножество. Возвращает ссылку на вызывающий массив
`vallaray<T> &operator=(const mask_array<T> &ob);`	Присваивает подмножество. Возвращает ссылку на вызывающий массив
`vallaray<T> &operator=(const indirect_array<T> &ob);`	Присваивает подмножество. Возвращает ссылку на вызывающий массив
`vallaray<T> operator+()const;`	Применяет унарный плюс ко всем элементам вызывающего массива. Возвращает результирующий массив
`vallaray<T> operator-()const;`	Применяет унарный минус ко всем элементам вызывающего массива. Возвращает результирующий массив
`vallaray<T> operator~()const;`	Применяет побитовую операцию отрицания ко всем элементам вызывающего массива. Возвращает результирующий массив
`vallaray<T> operator!()const;`	Применяет логическую операцию отрицания ко всем элементам вызывающего массива. Возвращает результирующий массив
`vallaray<T> &operator+=(const T &v)const;`	Прибавляет величину *v* к каждому элементу вызывающего массива. Возвращает результирующий массив
`vallaray<T> &operator-=(const T &v)const;`	Вычитает величину *v* из каждого элемента вызывающего массива. Возвращает результирующий массив
`vallaray<T> &operator/=(const T &v)const;`	Делит каждый элемент вызывающего массива на величину *v*. Возвращает результирующий массив
`vallaray<T> &operator*=(const T &v)const;`	Умножает каждый элемент вызывающего массива на величину *v*. Возвращает результирующий массив
`vallaray<T> &operator%=(const T &v)const;`	Присваивает каждому элементу вызывающего массива остаток от деления на величину *v*. Возвращает результирующий массив
`vallaray<T> operator^=(const T &v)const;`	Применяет логическую операцию исключающего "ИЛИ" ко всем элементам вызывающего массива и величине *v*. Возвращает результирующий массив
`vallaray<T> operator&=(const T &v)const;`	Применяет логическую операцию исключающего "И" ко всем элементам вызывающего массива и величине *v*. Возвращает результирующий массив
`vallaray<T> operator|=(const T &v)const;`	Применяет логическую операцию "ИЛИ" ко всем элементам вызывающего массива и величине *v*. Возвращает результирующий массив
`vallaray<T> &operator<<=(const T &v)const;`	Применяет операцию побитового сдвига влево на величину *v* ко всем элементам вызывающего массива. Возвращает результирующий массив
`vallaray<T> &operator>>=(const T &v)const;`	Применяет операцию побитового сдвига вправо на величину *v* ко всем элементам вызывающего массива. Возвращает результирующий массив

`vallaray<T> &operator+=(const vallaray<T> &ob)const;`	Складывает соответствующие элементы вызывающего массива и объекта *ob*. Возвращает результирующий массив
`vallaray<T> &operator-=(const vallaray<T> &ob)const;`	Вычитает объект *ob* из соответствующих элементов вызывающего массива. Возвращает результирующий массив
`vallaray<T> &operator/=(const vallaray<T> &ob)const;`	Делит элементы вызывающего массива на соответствующие элементы объекта *ob*. Возвращает результирующий массив
`vallaray<T> &operator*=(const vallaray<T> &ob)const;`	Умножает элементы вызывающего массива на соответствующие элементы объекта *ob*. Возвращает результирующий массив
`vallaray<T> &operator%=(const vallaray<T> &ob)const;`	Присваивает каждому элементу вызывающего массива остаток от деления на соответствующий элемент массива *ob*. Возвращает результирующий массив
`vallaray<T> & operator^=(const vallaray<T> &ob)const;`	Применяет логическую операцию исключающего "ИЛИ" ко всем элементам вызывающего массива и соответствующим элементам объекта *ob*. Возвращает результирующий массив
`vallaray<T> & operator&=(const vallaray<T> &ob)const;`	Применяет логическую операцию "И" ко всем элементам вызывающего массива и соответствующим элементам объекта *ob*. Возвращает результирующий массив
`vallaray<T> & operator|=(const vallaray<T> &ob)const;`	Применяет логическую операцию "ИЛИ" ко всем элементам вызывающего массива и соответствующим элементам объекта *ob*. Возвращавт результирующий массив
`vallaray<T> &operator<<=(const vallaray<T> &ob)const;`	Применяет операцию побитового сдвига влево на величину, определенную соответствующим элементом объекта *ob*, ко всем элементам вызыввющего массива. Возвращает результирующий массив
`vallaray<T> &operator>>=(const vallaray<T> &ob)const;`	Применяет операцию побитового сдвига вправо на величину, определенную соответствующим элементом объекта *ob*, ко всем элементам вызывающего мвссива. Возвращает результирующий массив
`T &operator[](size_t indx);`	Возвращает ссылку на элемент с индексом *indx*
`T operator[](size_t indx) const;`	Возвращает значение элемента с индексом *indx*
`slice_array<T> operator[](slice ob);`	Возвращает указанное подмножество
`valarray<T> operator[](slice &ob) const;`	Возвращает указанное подмножество
`gslice_array<T> operator[](const gslice &ob);`	Возвращает указанное подмножество
`mask_array<T> operator[](valarray<bool> &ob);`	Возвращавт указанное подмножество
`valarray<T> operator[](valarray<bool> &ob) const;`	Возвращает указанное подмножество
`indirect_array<T> operator[](const valarray<size_t> &ob);`	Возвращает указанное подмножество
`valarray<T> operator[](const valarray<size_t> &ob) const;`	Возвращает указанное подмножество

`void resize(size_t num, T v = T());`	Изменяет размер вызывающего массива. Если массив нужно увеличить, добавляемым элементам присваивается значение *v*
`size_t size() const;`	Возвращает размер (количество элементов) вызывающего массива
`vallaray<T> shift(int num) const;`	Сдвигает вызывающий массив влево на *num* позиций. Возвращает массив, содержащий результат операции
`T sum() const;`	Возвращает сумму значений, хранящихся в вызывающем массиве

Таблица 37.3. Функции, определенные для класса valarray

`template<class T> valarray<T> operator+(const valarray<T> ob, const T &v);`	Добавляет значение *v* к каждому элементу объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator+(const T &v, const valarray<T> ob);`	Добавляет значение *v* к каждому элементу объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valaray<T> operator+(const vallaray<T> ob1, const vallaray<T> &ob2);`	Складывает каждый элемент объекта *ob1* с соответствующим элементом объекта *ob2*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator-(const valarray<T> ob, const T &v);`	Вычитает значение *v* из каждого элемента объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator-(const T &v, const valarray<T> ob);`	Вычитает каждый элемент объекта *ob* из величины *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator-(const valarray<T> ob1 const valarray<T> &ob2);`	Вычитает каждый элемент объекта *ob2* из соответствующего элемента объекта *ob1*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator*(const valarray<T> ob, const T &v);`	Умножает значение *v* на каждый элемент объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator*(const T &v, const valarray<T> ob);`	Умножает значение *v* на каждый элемент объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator-(const valarray<T> ob1, const valarray<T> &ob2);`	Перемножает соответствующие элементы объектов *ob2* и *ob1*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator/(const valarray<T> ob, const T &v);`	Делит каждый элемент объекта *ob* на величину *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator/(const T &v, const valarray<T> ob);`	Делит величину *v* на каждый элемент объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator/(const valarray<T> ob1, const valarray<T> &ob2);`	Делит каждый элемент объекта *ob1* на соответствующий элемент объекта *ob1*. Возвращает массив, содержащий результат операции

`template<class T> valarray<T> operator%(const valarray<T> ob, const T &v);`	Вычисляет остаток от деления каждого элемента объекта *ob* на величину *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator%(const T &v, const valarray<T> ob);`	Вычисляет остаток от деления величины *v* на каждый элемент объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator%(const valaray<T> ob1, const valaray<T> &ob2);`	Вычисляет остаток от деления каждого элемента объекта *ob1* на соответствующий элемент объекта *ob2*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator^(const valarray<T> ob, const T &v);`	Применяет операцию исключающего "ИЛИ" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator^(const T &v, const valarray<T> 'ob);`	Применяет операцию исключающего "ИЛИ" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator^(const valarray<T> ob1 const valarray<T> &ob2);`	Применяет операцию исключающего "ИЛИ" к соответствующим элементам объектов *ob1* и *ob2*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator&(const valarray<T> ob, const T &v);`	Применяет операцию "И" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator&(const T &v, const valarray<T> ob);`	Применяет операцию "И" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator&(const valarray<T> ob1, const valarray<T> &ob2);`	Применяет операцию "И" к соответствующим элементам объектов *ob1* и *ob2*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator|(const valarray<T> ob, const T &v);`	Применяет операцию "ИЛИ" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator|(const T &v, const valarray<T> ob);`	Применяет операцию "ИЛИ" к каждому элементу объекта *ob* и величине *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator|(const valarray<T> ob1, const valarray<T> &ob2);`	Применяет операцию "ИЛИ" к соответствующим элементам объектов *ob1* и *ob2*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator<<(const valarray<T> ob, const T &v);`	Применяет операцию побитового сдвига влево на величину *v* к каждому элементу объекта *ob*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator<<(const T &v, const vallaray<T> ob);`	Применяет операцию побитового сдвига влево на величину, определенную соответствующим элементом объекта *ob*, к значению *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator<<(const valarray<T> ob1, const valarray<T> &ob2);`	Применяет операцию побитового сдвига влево на величину, определенную соответствующим элементом объекта *ob2*, к каждому элементу объекта *ob1*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator>>(const valarray<T> ob, const T &v);`	Выполняет операцию побитового сдвига вправо на величину *v* к каждому элементу объекта *ob*. Возвращает массив, содержащий результат операции

`template<class T> valarray<T> operator>>(const T &v, const valarray<T> ob);`	Применяет операцию побитового сдвига вправо на величину, определенную соответствующим элементом объекта *ob*, к значению *v*. Возвращает массив, содержащий результат операции
`template<class T> valarray<T> operator>>(const valarray<T> ob1, const valarray<T> &ob2);`	Применяет операцию побитового сдвига вправо на величину, определенную соответствующим элементом объекта *ob2*, к каждому элементу объекта *ob1*. Возвращает массив, содержащий результат операции
`template<class T> valarray<bool> operator==(const valarray<T> ob, const T &v);`	Выполняет сравнение *ob[i]==v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator==(const T &v, const valarray<T> ob);`	Выполняет сравнение *v==ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator==(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет сравнение *ob1[i]==ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator!=(const valarray<T> ob, const T &v);`	Выполняет сравнение *ob[i]!=v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator!=(const T &v, const valarray<T> ob);`	Выполняет сравнение *v!=ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator!=(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет сравнение *ob1[i]!=ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<(const valarray<T> ob, const T &v);`	Выполняет сравнение *ob[i]<v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<(const T &v, const vallaray<T> ob);`	Выполняет сравнение *v<ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет сравнение *ob1[i]<ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<=(const valarray<T> ob, const T &v);`	Выполняет сравнение *ob[i]<=v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<=(const T &v, const vallaray<T> ob);`	Выполняет сравнение *v<=ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator<=(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет сравнение *ob1[i]<=ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator>(const valarray<T> ob, const T &v);`	Выполняет сравнение *ob[i]>v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator>(const T &v, const vallaray<T> ob);`	Выполняет сравнение *v>ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат

`template<class T> valaray<bool> operator>(const vallaray<T> ob1, const vallaray<T> &ob2);`	Выполняет сравнение *ob1[i]>ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valaray<bool> operator>=(const vallaray<T> ob, const T &v);`	Выполняет сравнение *ob[i]>=v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator>=(const T &v, const valarray<T> ob);`	Выполняет сравнение *v>=ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator>=(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет сравнение *ob1[i]&>=ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator&&(const valarray<T> ob, const T &v);`	Выполняет операцию *ob[i]&&v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator&&(const T &v, const valarray<T> ob);`	Выполняет операцию *v&&ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator&&(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет операцию *ob1[i]&&ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator||(const valarray<T> ob, const T &v);`	Выполняет операцию *ob[i]||v* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<bool> operator||(const T &v, const valarray<T> ob);`	Выполняет операцию *v||ob[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат
`template<class T> valarray<T> operator||(const valarray<T> ob1, const valarray<T> &ob2);`	Выполняет операцию *ob1[i]||ob2[i]* для каждого значения *i*. Возвращает массив булевых значений, содержащий результат

Таблица 37.4. Трансцендентные функции, определенные для класса valarray

`template<class T> valarray<T> abs(const valarray<T> &ob);`	Вычисляет абсолютное значение каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> acos(const valarray<T> &ob);`	Вычисляет арккосинус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> acos(const valarray<T> &ob);`	Вычисляет арксинус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> atan(const valarray<T> &ob);`	Вычисляет арктангенс каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> atan2(const valarray<T> &ob1, const valarray<T> &ob2);`	Вычисляет арктангенс величин *ob1[i]/ob2[i]*. Возвращает массив, содержащий результат
`template<class T> valarray<T> atan2(const T &v, const valarray<T> &ob);`	Вычисляет арктангенс величин *v/ob[i]*. Возвращает массив, содержащий результат

`template<class T> valarray<T> atan2(const valarray<T> &ob, const T &v);`	Вычисляет арктангенс величины *ob[i]/v* каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> cos(const valarray<T> &ob);`	Вычисляет косинус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> cosh(const valarray<T> &ob);`	Вычисляет гиперболический косинус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> exp(const valarray<T> &ob);`	Вычисляет экспоненту каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> log(const valarray<T> &ob);`	Вычисляет натуральный логарифм каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> log10(const valarray<T> &ob);`	Вычисляет десятичный логарифм каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> pow(const valarray<T> &ob1, const valarray<T> &ob2);`	Вычисляет величину $ob1[i]^{ob2[i]}$ для всех значений *i*. Возвращает массив, содержащий результат
`template<class T> valarray<T> pow(const T &v, const valarray<T> &ob);`	Вычисляет величину $v^{ob[i]}$ для всех значений *i*. Возвращает массив, содержащий результат
`template<class T> valarray<T> pow(const valarray<T> &ob, const T &v);`	Вычисляет величину $ob[i]^{v}$ для всех значений *i*. Возвращает массив, содержащий результат
`template<class T> valarray<T> sin(const valarray<T> &ob);`	Вычисляет синус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> sinh(const valarray<T> &ob);`	Вычисляет гиперболический синус каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> sqrt(const valarray<T> &ob);`	Вычисляет квадратный корень каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> tan(const valarray<T> &ob);`	Вычисляет тангенс каждого элемента объекта *ob*. Возвращает массив, содержащий результат
`template<class T> valarray<T> tanh(const valarray<T> &ob);`	Вычисляет гиперболический тангенс каждого элемента объекта *ob*. Возвращает массив, содержащий результат

Применение класса **valarray** иллюстрируется следующей программой.

```
// Демонстрация применения класса valarray.
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;

int main()
{
  valarray<int> v(10);
  int i;

  for(i=0; i<10; i++) v[i] = i;
```

```
  cout << "Исходное содержимое: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  v = v.cshift(3);

  cout << "Сдвинутое содержимое: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  valarray<bool> vb = v < 5;
  cout << "Эти элементы меньше 5: ";
  for(i=0; i<10; i++)
    cout << vb[i] << " ";
  cout << endl << endl;

  valarray<double> fv(5);
  for(i=0; i<5; i++) fv[i] = (double) i;

  cout << "исходное содержимое: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = sqrt(fv);

  cout << "Квадратные корни: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv + fv;
  cout << "Удвоенные квадратные корни: ";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  fv = fv - 10.0;
  cout << "После вычитания числа 10 из каждого элемента:\n";
  for(i=0; i<5; i++)
    cout << fv[i] << " ";
  cout << endl;

  return 0;
}
```

Результат работы этой программы выглядит так.

```
Исходное содержимое: 0 1 2 3 4 5 6 7 8 9
Сдвинутое содержимое: 3 4 5 6 7 8 9 0 1 2
Эти элементы меньше 5: 1 1 0 0 0 0 0 1 1 1

Исходное содержимое: 0 1 2 3 4
Квадратные корни: 0 1 1.41421 1.73205 2
```

```
Удвоенные квадратные корни: 0 2 2.82843 3.4641 4
После вычитания числа 10 из каждого элемента:
-10 -8 -7.17157 -6.5359 -6
```

Классы slice и gslice

В заголовке **<valarray>** определены два служебных класса: **slice** и **gslice**. Эти классы инкапсулируют сечение (т.е. часть) массива. Они используют операторную функцию **operator[]**, определенную для подмножеств типа **valarray**.

Класс **slice** выглядит следующим образом.

```
class slice {
public:
  slice();
  slice(size_t start, size_t len, size_t interval);
  size_t start() const;
  size_t size() const;
  size_t stride();
};
```

Первый конструктор создает пустое сечение. Второй конструктор создает сечение по заданным первому элементу, длине и величине интервала между элементами (т.е. по *шагу индекса* (stride)). Эти значения вычисляются с помощью функций-членов.

Рассмотрим программу, демонстрирующую применение класса **slice**.

```
// Демонстрация класса slice.
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(10), result;
  int i;

  for(i=0; i<10; i++) v[i] = i;

  cout << "Содержимое массива v: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[slice(0,5,2)];

  cout << "Результат: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

Результаты работы этой программы приведены ниже.

```
Содержимое массива v: 0 1 2 3 4 5 6 7 8 9
Результат: 0 2 4 6 8
```

Как видим, результирующий массив состоит из пяти элементов массива **v**, начиная с 0, причем разность между индексами соседних элементов равна 2.

Класс **gslice** состоит из следующих членов.

```
class gslice {
public:
   gslice();
   gslice()(size_t start, const vallaray<size_t> &len,
            const vallaray<size_t> &intervals);
   size_t start() const;
   vallaray<size_t> size() const;
   valarray<size_t> stride() const;
};
```

Первый конструктор создает пустое сечение. Второй конструктор создает сечение по заданным первому элементу, длине и величине интервала между элементами (т.е. по *шагу индекса* (stride)). Количество элементов массивов **lens** и **intervals** должно совпадать. Эти значения вычисляются с помощью функций-членов. Класс **gslice** применяется для создания многомерных массивов на основе объекта класса **valarray**, который по определению всегда является одномерным.

Следующая программа демонстрирует применение класса **gslice**.

```
// Демонстрация класса gslice.
#include <iostream>
#include <valarray>
using namespace std;

int main()
{
  valarray<int> v(12), result;
  valarray<size_t> len(2), interval(2);
  int i;

  for(i=0; i<12; i++) v[i] = i;

  len[0] = 3; len[1] = 3;
  interval[0] = 2; interval[1] = 3;

  cout << "Содержимое массива v: ";
  for(i=0; i<12; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[gslice(0,len,interval)];

  cout << "Результат: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

Результат работы этой программы выглядит следующим образом.

```
Содержимое массива: 0 1 2 3 4 5 6 7 8 9 10 11
Результат: 0 3 6 2 5 8 4 7 10
```

Вспомогательные классы

Числовые классы используют вспомогательные классы **slice_array**, **gslice_array**, **indirect_array** и **mask_array**, которые никогда не конкретизируются явно.

Числовые алгоритмы

Заголовок **<numeric>** содержит определения четырех числовых алгоритмов, предназначенных для обработки содержимого контейнера. Рассмотрим каждый из них.

Класс accumulate

Алгоритм **accumulate()** вычисляет сумму всех элементов указанного диапазона и возвращает результат. Его прототип выглядит следующим образом.

```
template <class InIter, class T>
  T accumulate(InIter start, InIter end, T v);
template <class InIter, class T, class BinFunc>
  T accumulate(InIter start, InIter end, T v, BinFunc func);
```

Здесь класс **T** описывает тип значений, которыми оперирует алгоритм. Первая версия алгоритма вычисляет сумму всех элементов диапазона, ограниченного итераторами *start* и *end*. Функция *func* во второй версии определяет порядок суммирования элементов. Параметр *v* задает начальное значение, с которого начинается суммирование.

Рассмотрим программу, иллюстрирующую алгоритм **accumulate**.

```
// Демонстрация алгоритма accumulate.
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(5);
  int i, total;

  for(i=0; i<5; i++) v[i] = i;

  total = accumulate(v.begin(), v.end(), 0);

  cout << "Сумма элементов массива v: " << total;

  return 0;
}
```

Результат работы программы приведен ниже.

```
Сумма элементов массива v: 10
```

Алгоритм adjacent_difference

Алгоритм **adjacent_difference()** создает новую последовательность, каждый элемент которой равен разности между соседними элементами исходной последовательности. (Первый элемент в результирующей последовательности равен первому элементу исходного массива.) Прототипы алгоритма **adjacent_difference()** приведены ниже.

```
template <class InIter, class OutIter>
  OutIter adjacent_difference(InIter start,
                              InIter end, OutIter result);
```

```
template <class InIter, class OutIter, class BinFunc>
  OutIter adjacent_difference(InIter start, InIter end,
                              OutIter result, BinFunc func);
```

Здесь параметры *start* и *end* являются итераторами, установленными на начало и конец исходной последовательности. Результирующая последовательность хранится в объекте, на который ссылается итератор *resut*. Первая версия алгоритма вычисляет разность между соседними элементами, вычитая элемент, занимающий *n*-ю позицию, из элемента, занимающего *n*+1-ю позицию. Во второй версии к каждой паре соседних элементов применяется бинарная функция *func*. Алгоритм возвращает итератор *result*.

Рассмотрим пример применения алгоритма **adjacent_difference()**.

```
// Демонстрация алгоритма adjacent_difference()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v(10), r(10);
  int i;

  for(i=0; i<10; i++) v[i] = i*2;
  cout << "Исходная последовательность: ";
  for(i=0; i<10; i++)
    cout << v[i] << " ";
  cout << endl;

  adjacent_difference(v.begin(), v.end(), r.begin());

  cout << "Результат: ";
  for(i=0; i<10; i++)
    cout << r[i] << " ";

  return 0;
}
```

Результат работы программы приведен ниже.

```
Исходная последовательность: 0 2 4 6 8 10 12 14 16 18
Результат: 0 2 2 2 2 2 2 2 2 2
```

Как видим, результирующая последовательность содержит разности между значениями соседних элементов.

Алгоритм inner_product

Алгоритм **inner_product()** вычисляет скалярное произведение двух последовательностей и возвращает результат. Он имеет следующие прототипы.

```
template <class InIter1, class InIter2, class T>
  T inner_product(InIter1 start1, InIter1 end1,
                  InIter2 start2, T v);
template <class InIter1, class InIter2, class T,
          class BinFunc1, class BinFunc2>
  T inner_product(InIter1 start1, InIter1 end1, InIter2 start2, T v,
                  BinFunc1 func1, BinFunc2 func2);
```

Здесь параметры *start1* и *end1* являются итераторами, установленными на начало и конец первой последовательности. Итератор *start2* установлен на начало второй последовательности. Параметр *v* задает значение, с которого начинается суммирование. Во второй версии бинарная функция *func1* определяет способ суммирования, а бинарная функция *func2* — способ перемножения элементов последовательностей.

Рассмотрим программу, демонстрирующую алгоритм **inner_product()**.

```
// Демонстрация алгоритма inner_product()
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
  vector<int> v1(5), v2(5);
  int i, total;

  for(i=0; i<5; i++) v1[i] = i;
  for(i=0; i<5; i++) v2[i] = i+2;

  total = inner_product(v1.begin(), v1.end(),
                        v2.begin(), 0);

  cout << "Скалярное произведение: " << total;

  return 0;
}
```

Результат работы программы приведен ниже.

```
Скалярное произведение: 50
```

Алгоритм partial_sum

Алгоритм **partial_sum()** суммирует значения элементов последовательности, записывая текущую сумму в новую последовательность. (Иначе говоря, он создает последовательность, в которой хранятся частичные суммы элементов исходной последовательности.) Первый элемент результата совпадает с первым элементом исходной последовательности. Прототипы алгоритмов имеют следующий вид.

```
template <class InIter, class OutIter>
  OutIter partial_sum(InIter start, InIter end, OutIter result);
template <class InIter, class OutIter, class BinFunc>
  OutIter partial_sum(InIter start, InIter end,
                      OutIter result, BinFunc func);
```

Здесь параметры *start* и *end* являются итераторами, установленными на начало и конец исходной последовательности. Результирующая последовательность хранится в объекте, на который ссылается итератор *resut*. Во второй версии способ суммирования определяется бинарной функцией *func*. Алгоритм возвращает итератор *result*.

Рассмотрим пример, демонстрирующий применение алгоритма **partial_sum()**.

```
// Демонстрация алгоритма partial_sum().
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;
```

```
int main()
{
  vector<int> v(5), r(5);
  int i;

  for(i=0; i<10; i++) v[i] = i;
  cout << "Исходная последовательность: ";
  for(i=0; i<5; i++)
    cout << v[i] << " ";
  cout << endl;

  partial_sum(v.begin(), v.end(), r.begin());

  cout << "Результат: ";
  for(i=0; i<5; i++)
    cout << r[i] << " ";

  return 0;
}
```

Результат работы программы приведен ниже.

```
Исходная последовательность: 0 1 2 3 4
Результат: 0 1 3 6 10
```

Полный справочник по

Глава 38

Обработка исключительных ситуаций и прочие классы

В этой главе описываются классы, предназначенные для обработки исключительных ситуаций. Кроме того, рассматриваются классы **auto_ptr** и **pair**, а также приводится краткий обзор библиотеки локализации.

Исключительные ситуации

Стандарт языка C++ предусматривает два заголовка, связанных с исключительными ситуациями: **<exception>** и **<stdexcept>**. Исключительные ситуации предназначены для выявления ошибок. Рассмотрим каждый из упомянутых заголовков.

Заголовок <exception>

В заголовке **<exception>** определены классы, типы и функции, связанные с обработкой исключительных ситуаций. Классы, определенные в заголовке **<exception>**, приводятся ниже.

```
class exception {
public:
  exception() throw();
  exception(const bad_exception &ob) throw();
  virtual ~exception() throw();

  exception &operator=(const exception &ob) throw();
  virtual const char *what() const throw();
};

class bad_exception: public exception {
public:
  bad_exception() throw();
  bad_exception(const bad_exception &ob) throw();
  virtual ~bad_exception() throw();

  bad_exception &operator=(const bad_exception &ob) throw();
  virtual const char *what() const throw();
};
```

Класс **exception** является базовым для всех исключительных ситуаций, определенных в стандарте языка C++. Класс **bad_exception** описывает тип исключительной ситуации, генерируемой функцией **unexpected()**. В каждом классе есть функция-член **what()**, возвращающая указатель на строку, которая завершается нулевым байтом и описывает возникшую исключительную ситуацию.

Класс **exception** является базовым для нескольких важных классов. К ним относятся: 1) класс **bad_alloc**, генерирующий исключительную ситуацию при неудачном выполнении оператора **new**; 2) класс **bad_typeid**, генерирующий исключительную ситуацию при неправильном выполнении оператора **typeid**; и 3) класс **bad_cast**, генерирующий исключительную ситуацию при некорректном динамическом приведении типов. Эти классы состоят из тех же членов, что и класс **exception**.

В заголовке **<exception>** определены следующие типы.

Тип	*Прототип*
terminate_handler	**void(*terminate_handler)();**
unexpected_handler	**void(*unexpected_handler)();**

Функции, объявленные в заголовке <exception>, перечислены в табл. 38.1.

Таблица 38.1. Функции, определенные в заголовке <exception>

`terminate_handler set_terminate(terminate_handler fn) throw();`	Назначает функцию, заданную параметром *fn*, обработчиком неисправимых ошибок. Возвращает указатель на старый обработчик этого вида
`terminate_handler set_unexpected(unexpected_handler fn) throw();`	Назначает функцию, заданную параметром *fn*, обработчиком непредвиденных исключительных ситуаций. Возвращает указатель на старый обработчик этого вида
`void terminate();`	Вызывает обработчик неперехваченной и неисправимой ошибки. По умолчанию вызывает функцию `abort()`
`bool uncaught_exception();`	Возвращает значение `true`, если исключительная ситуация не была перехвачена
`void unexpected();`	Вызывает обработчик непредвиденной исключительной ситуации, если функция сгенерировала непредусмотренную исключительную ситуацию. По умолчанию вызывает функцию `terminate()`

Заголовок <stdexcept>

В заголовке `<stdexcept>` определены несколько стандартных исключительных ситуаций, генерируемых библиотечными функциями или системой поддержки выполнения программ (run-time system). В заголовке `<stdexcept>` определены два вида исключительных ситуаций: логические ошибки и ошибки, возникающие при выполнении программ (run-time errors). Логические ошибки — результат неправильного программирования. Ошибки, возникающие при выполнении программ, являются следствием неверной работы библиотечных функций или операционной системы и не могут контролироваться программистом.

Стандартные исключительные ситуации, порождаемые логическими ошибками, описываются классами, производными от класса `logic_error`.

Исключительная ситуация	*Смысл*
`domain_error`	Выход за пределы допустимого диапазона значений.
`invalid_argument`	При вызове функции указан неверный аргумент.
`length_error`	Попытка создать слишком крупный объект.
`out_of_range`	Аргумент функции выходит за пределы допустимого диапазона значений.

Исключительные ситуации, перечисленные ниже, являются производными от базового класса `runtime_error`.

Исключительная ситуация	*Смысл*
`overflow_error`	Переполнение при выполнении арифметических операций.
`range_error`	Выход за пределы допустимого диапазона.
`underflow_error`	Потеря значимости.

Класс auto_ptr

Класс **auto_ptr**, объявленный в заголовке **<memory>**, весьма интересен. Объект этого класса является указателем, имеющим право владения объектом, на который он ссылается. Это право можно передать другому объекту класса **auto_ptr**, однако некоторые объекты владеют объектами вечно. Эта схема гарантирует правильное уничтожение динамических объектов при любых обстоятельствах с помощью деструкторов самих объектов. Например, когда один объект класса **auto_ptr** присваивается другому, право владения объекта переходит левому операнду оператора присваивания. При уничтожении указателей объект, на который они ссылаются, уничтожается лишь один раз, а именно: когда уничтожается указатель, имеющий право владения данным объектом. Одно из преимуществ такого подхода заключается в том, что он позволяет уничтожать динамические объекты в ходе обработки исключительных ситуаций.

Шаблонная спецификация класса **auto_ptr** приведена ниже.

```
template <class T> class auto_ptr;
```

Здесь класс **T** представляет собой тип указателя, хранящегося в объекте класса **auto_ptr**.

Класс **auto_ptr** имеет следующие конструкторы.

```
explicit auto_ptr(T* ptr = 0) throw();
auto_ptr(auto_ptr &ob) throw();
template <class T2> auto_ptr(auto_ptr<T2> &ob) throw();
```

Первый конструктор создает объект класса **auto_ptr**, ссылающийся на объект, заданный параметром *ptr*. Второй конструктор создает копию объекта класса **auto_ptr**, заданного параметром *ob*, и передает новому объекту право владения. Третий конструктор преобразует объект *ob* в экземпляр класса **T** (если этой возможно) и передает ему право владения.

В классе **auto_ptr** определены операции **=**, ***** и **->**. Кроме того, он содержит две функции-члена.

```
T *get() const throw();
T *release() const throw();
```

Функция **get()** возвращает указатель на объект класса **T**. Функция **release()** лишает вызывающий объект класса **auto_ptr** права собственности и возвращает указатель на объект класса **T**. После вызова функции **release()** объект класса **T**, которым владеет объект класса **auto_ptr**, вышедший за пределы видимости, не уничтожается.

Рассмотрим программу, демонстрирующую применение класса **auto_ptr**.

```
// Демонстрация класса auto_ptr.
#include <iostream>
#include <memory>
using namespace std;

class X {
public:
  X() { cout << "создание\n"; }
  ~X() { cout << "уничтожение\n"; }
  void f() { cout << "Внутри функции f()\n"; }
};

int main()
{
```

```
  auto_ptr<X> p1(new X), p2;

  p2 = p1; // Передача права владения
  p2->f();

  // Объекты класса auto_ptr можно присваивать
  // обычным указателям.
  X *ptr = p2.get();
  ptr->f();

  return 0;
}
```

Результат работы программы показан ниже.

```
создание
Внутри функции f()
Внутри функции f()
уничтожение
```

Обратите внимание на то, что функцию **f()**, являющуюся членом объекта **X**, можно вызывать как с помощью объекта класса **auto_ptr**, так и через обычный указатель, возвращаемый функцией **get()**.

Класс pair

Класс **pair** используется для хранения *пар* объектов, как это происходит в ассоциативном контейнере. Его шаблонная спецификация имеет следующий вид.

```
template <class Ktype, class Vtype> struct pair {
  typedef Ktype first_type;
  typedef Vtype second_type;
  Ktype first;
  Vtype second;

  // Конструкторы
  pair();
  pair(const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

В поле **first** обычно хранится ключ, а в поле **value** — значение, связанное с этим ключом.

В классе **pair** определены следующие операции: **==**, **!=**, **<**, **<=**, **>** и **>=**.

Пары можно создавать либо с помощью конструктора класса **pair**, либо с помощью функции **make_pair()**, которая объединяет свои параметры. Функция **make_pair()** является обобщенной. Ее прототип выглядит следующим образом.

```
template <class Ktype, class Vtype>
  pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

Как видим, эта функция возвращает пару, состоящую из объектов, имеющих типы *Ktype* и *Vtype*. Преимущество функции **make_type()** заключается в том, что типы объектов, из которых образуется пара, определяются компилятором автоматически, а не задаются программистом явным образом.

Класс **pair** и функция **make_pair()** используют заголовок **<utility>**.

Локализация

Стандарт языка C++ содержит большую библиотеку классов, предназначенных для локализации программ. Эти классы позволяют приложениям задавать или получать информацию о геополитическом окружении, в котором они выполняются. Таким образом, они определяют, например, формат представления денежных величин, времени и даты, а также порядок их сравнения. Кроме того, эти классы позволяют классифицировать символы. Библиотека локализации использует заголовок **<locale>**. Она оперирует наборами классов, в которых определяются *аспекты локализации* (facets) — информация, связанная с локализацией программ. Все аспекты локализации являются производными от класса **facet**, который в свою очередь вложен в класс **locale**.

Честно говоря, библиотека локализации крайне велика и сложна. Ее описание выходит за рамки нашей книги. Хотя большинство программистов не применяют библиотек локализации непосредственно, тот специалист, который занимается международными проектами, должен внимательно изучить ее особенности.

Прочие классы

В стандартной библиотеке языка C++ определены еще несколько довольно интересных классов.

Класс	*Описание*
type_info	Используется в сочетании с оператором **typeid**, описанным в главе 22. Использует заголовок **<typeinfo>**.
numeric_limits	Инкапсулирует пределы изменения чисел. Использует заголовок **<limits>**.
raw_storage_iterator	Инкапсулирует средства распределения неинициализированной памяти. Использует заголовок **<memory>**.

Полный справочник по

Часть V

Приложения на языке C++

В этой части книги рассматриваются два простых приложения. Преследуется двоякая цель: во-первых, приведенные примеры иллюстрируют преимущества объектно-ориентированного программирования, а во-вторых, они демонстрируют, как с помощью языка C++ можно решить две совершенно разные задачи.

Полный справочник по C++

Глава 39

Интеграция новых классов: пользовательский класс для работы со строками

В этой главе описывается процесс разработки и реализации небольшого класса, обеспечивающего работу со строками. Как известно, стандарт языка C++ содержит полноценный мощный класс для работы со строками под названием **basic_string**. Цель этой главы заключается не в разработке альтернативного класса. На конкретном примере мы стремимся дать читателю ясное представление о разработке и интеграции новых типов данных. Создание класса для работы со строками является весьма ярким примером. В прошлом многие программисты оттачивали мастерство, создавая свои собственные строковые классы. В этой главе мы займемся тем же.

Хотя класс, который мы будем разрабатывать, намного проще стандартного, у нас будет одно преимущество: мы с самого начала можем полностью контролировать процесс реализации строки и манипуляции ими. В некоторых ситуациях это может оказаться полезным. В конце концов, это просто интересно!

Класс StrType

Наш класс намного проще стандартного. Разумеется, он не настолько велик и сложен. Перечислим требования, которым должен соответствовать наш класс.

- Строки должны присваиваться с помощью оператора присваивания.
- Строковым объектам необходимо присваивать как строковые объекты, так и строки, заключенные в кавычки.
- Конкатенация двух строк должна осуществляться с помощью оператора "+".
- Удаление подстроки должно осуществляться с помощью оператора "-".
- Сравнение строк должно выполняться с помощью операторов сравнения.
- Строковые объекты должны инициализироваться либо строками, заключенными в кавычки, либо другим строковым объектом.
- Строки должны иметь произвольную и переменную длину. Следовательно, они должны размещаться в динамической памяти.
- Следует предусмотреть способ преобразования строкового объекта в строку, завершающуюся нулевым байтом.

Хотя наш класс по определению слабее стандартного, он обладает свойством, которым не обладает класс **basic_string**: удаление подстроки с помощью оператора "-".

Класс, управляющий работой со строками, называется **StrType**. Его объявление приведено ниже.

```
class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // Конструктор копирования

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // Присваивание объекта
```

```
                                    // класса StrType.
  StrType operator=(char *s);       // Присваивание строки,
                                    // взятой в кавычки.

  StrType operator+(StrType &o); // Конкатенация двух объектов
                                    // класса StrType.
  StrType operator+(char *s);       // Конкатенация двух строк,
                                    // взятых в кавычки.
  friend StrType operator+(char *s, StrType &o);  /*
                                    // Конкатенация строки,
                                    // заключенной в кавычки,
                                    // с объектом класса StrType */

  StrType operator-(StrType &o); // Вычитание подстроки.
  StrType operator-(char *s);       // Вычитание подстроки,
                                    // взятой в кавычки.

  // Сравнение объектов класса StrType.
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); )
  int operator<(StrType &o)  { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o)  { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // Операции над объектами класса StrType и строками,
  // взятыми в кавычки.
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s)  { return strcmp(p, s) < 0; }
  int operator>(char *s)  { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // Размер строки.
  void makestr(char *s)
  { strcpy(s, p); } // Создать строку, взятую в кавычки.

  operator char *()
  { return p; }     // Преобразовать в тип char *.
};
```

Закрытый раздел класса **StrType** содержит только два члена: **p** и **size**. При создании строкового объекта динамическая память выделяется оператором **new**, а указатель на выделенную область записывается в указатель **p**. Строка, адресуемая указателем **p**, представляет собой обычный массив символов, завершающийся нулевым байтом. Размер строки записывается в поле **size**, хотя с технической точки зрения это не обязательно. Поскольку строка, на которую ссылается указатель **p**, завершается нулем, ее размер можно вычислять в любое время. Однако, как мы увидим, это значение используется членами класса **StrType** так часто, что повторяющиеся вызовы функции **strlen()** становятся неэффективными.

Устройство класса **StrType** рассматривается в следующих разделах.

```
// Без явной инициализации.
StrType::StrType() {
  size = 1; // Создаем место для нулевого символа.
  try {
```

```
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(p, "");
}

// Инициализация с помощью строки, взятой в кавычки.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // Создаем место для нулевого символа
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(p, str);
}

// Инициализация с помощью объекта класса StrType.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(p, o.p);
}
```

Когда объект класса **StrType** создается, но не инициализируется, ему присваивается нулевая строка. Хотя строка не может оставаться неопределенной, предположение, что объект класса **StrType** содержит корректную строку, завершающуюся нулевым байтом, упрощает работу некоторых функций-членов.

Когда объект класса **StrType** инициализируется строкой, взятой в кавычки, сначала определяется размер строки. Затем с помощью оператора **new** выделяется достаточный объект памяти, и строка инициализации копируется в область памяти, адресуемую указателем **p**.

Процедура инициализации объекта класса **StrType** другим объектом этого типа практически совпадает с процедурой инициализации объекта строкой, взятой в кавычки. Этот вариант конструктора класса **StrType** является конструктором копирования. Он вызывается каждый раз, когда создаваемый объект инициализируется другим объектом класса **StrType**. Это значит, что он вызывается как при создании временных объектов, так и при передаче объектов типа **StrType** в качестве параметров функции. (Конструкторы копирования рассматривались в главе 14.)

Таким образом, следующие три объявления являются вполне допустимыми.

```
StrType x("my string"); // Применяется строка, взятая в кавычки.
StrType y(x); // Используется другой объект типа StrType.
StrType z; // Явная инициализация не выполняется.
```

Деструктор класса **StrType** просто освобождает память, на которую ссылается указатель **p**.

Ввод и вывод строк

Поскольку строки часто вводятся и выводятся, в классе **StrType** предусмотрены перегруженные версии операторов "<<" и ">>".

```
// Вывод строки.
ostream &operator<<(ostream &stream, StrType &o)
{
  stream << o.p;
  return stream;
}

// Ввод строки.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // Размер выбран произвольно -
               // при желании его можно изменить.
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}
```

Как видим, операция вывода строки очень проста. Однако следует обратить внимание, что параметр **o** передается по ссылке. Поскольку объекты класса **StrType** довольно велики, их эффективнее передавать по ссылке, а не по значению. По этой причине все параметры, имеющие тип **StrType** передаются только по ссылке. (Разрабатывая функцию, параметрами которой являются объекты класса **StrType**, следует иметь в виду это обстоятельство.)

Процедура ввода строки немного сложнее. Сначала строка считывается с помощью функции **getline()**. Длина наибольшей строки, которую можно ввести, ограничена 254 символами плюс нулевой байт, служащий признаком конца строки. Как указано в комментариях, эту величину можно изменить. Символы считываются до тех пор, пока не встретится символ перехода на новую строку. Если при вводе строки оказалось, что размер памяти, занятой объектом **o**, недостаточен, она освобождается, и строке выделяется другой участок памяти, имеющий более крупный размер. Затем в него копируется новая строка.

Функции присваивания

Объекты класса **StrType** можно присваивать двумя способами. Во-первых, объекту класса **StrType** можно присвоить другой объект этого типа. Во-вторых, ему можно

присвоить строку, взятую в кавычки. Для выполнения этих операций предназначены два варианта перегруженной операторной функции **operator=()**.

```
// Присвоение объекту класса StrType другого объекта этого типа.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete [] p; // Освобождаем память.
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Присвоение объекту класса StrType строки,
// взятой в кавычки.
 StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
  return *this;
}
```

Обе функции сначала проверяют, помещается ли присваиваемый объект в участке памяти, адресованном указателем **p**. Если нет, старый участок освобождается, и строке выделяется новая область памяти. Затем строка копируется в объект и функция возвращает результат. Таким образом, следующие виды присваивания являются вполне допустимыми.

```
StrType x("проверка"), y;

y = x; // Объекту класса StrType присваивается другой
       // объект класса StrType.

x = "новая строка для объекта x"; // Объекту класса StrType
             // присваивается строка, взятая в кавычки.
```

Каждая из функций присваивания возвращает присваиваемое значение (т.е. значение правого операнда). Это позволяет выполнять цепочки присваивания.

```
StrType x, y, z;

x = y = z = "проверка";
```

Конкатенация

Конкатенация двух строк выполняется с помощью оператора "+". Класс **StrType** предусматривает три вида конкатенации.

- Конкатенация объекта класса **StrType** с другим объектом этого типа.
- Конкатенация объекта класса **StrType** со строкой, взятой в кавычки.
- Конкатенация строки, взятой в кавычки, с объектом класса **StrType**.

В каждом из этих вариантов оператор "+" возвращает объект класса **StrType**, являющийся конкатенацией двух операндов. Ни один из этих операндов не модифицируется.

Рассмотрим перегруженную операторную функцию **operator+()**.

```
// Конкатенация двух объектов класса StrType.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Ошибка при выделении памяти\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}

// Конкатенация объекта класса StrType со строкой,
// взятой в кавычки.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
```

```
  strcpy(temp.p, p);

  strcat(temp.p, s);

  return temp;
}

// Конкатенация строки, взятой в кавычки,
// и объекта класса StrType.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}
```

Все три функции работают приблизительно одинаково. Во-первых, создается временный объект **temp** класса **StrType**. Этот объект хранит результат конкатенации и возвращается функциями в вызывающий модуль. Поскольку явная инициализация временного объекта не выполняется, при его создании выделяется только 1 байт (для символа-заполнителя). По этой причине память, на которую ссылается указатель **temp.p**, сначала освобождается, а затем функция выделяет область памяти, достаточную для того, чтобы в нее поместился результат конкатенации двух строк. В заключение, две строки копируются в область памяти, адресуемую указателем **temp.p**, и объект **temp** возвращается в вызывающий модуль.

Вычитание подстроки

В классе **basic_string** не предусмотрена такая полезная функция, как *вычитание подстроки*. В соответствии со своим определением в классе **StrType**, эта операция удаляет из объекта все подстроки, совпадающие с указанной подстрокой. Вычитание подстроки выполняется с помощью оператора "-".

Класс **StrType** предусматривает два вида вычитания подстроки. Первый вариант позволяет вычитать из объекта класса **StrType** другой объект этого типа. Второй вариант удаляет из объекта класса **StrType** все вхождения указанной подстроки, взятой в кавычки. Следующий фрагмент программы иллюстрирует оба варианта.

```
// Вычитание объектов класса StrType.
StrType StrType::operator-(StrType &substr)
{
```

```
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // Если символ не является первой
    temp.p[i] = *s1;     // буквой подстроки, копируем его
                         //в объект temp.
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) { // Если подстрока обнаружена,
                         // удаляем ее.
        s1 += j;
        i--;
      }
      else { // Если нет, продолжаем копирование.
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

// Вычитание из объекта класса StrType строки,
// взятой в кавычки.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // Если символ не является
                       // первой буквой подстроки,
      temp.p[i] = *s1; // копируем его в объект temp.
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // Если подстрока обнаружена,
                       // удаляем ее.
        s1 += j;
        i--;
      }
      else { // Если нет, продолжаем копирование.
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}
```

Обе функции копируют содержимое левого операнда в объект **temp**, удаляя любые вхождения подстроки, указанной в правом операнде. В качестве результата возвращается объект класса **StrType**. Ни один из операндов в ходе вычитания не изменяется.

Рассмотрим конкретные примеры вычитания подстрок.

```
StrType x("like C++"), y("like");
StrType z;

z = x - y; // Объект z будет содержать строку "I C++".

z = x - "C++"; // Объект z будет содержать строку "I like ".

// Удаление многократных вхождений подстроки.
z = "ABCDABCD";
x = z -"A"; // Объект x содержит строку "BCDBCD".
```

Операторы сравнения

Класс **StrType** предусматривает полный набор перегруженных операторов сравнения строк. Они перечислены в объявлении класса. Повторим их еще раз.

```
// Сравнение объектов класса StrType.
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o)  { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o)  { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // Операции над объектами класса StrType и строками,
  // взятыми в кавычки.
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s)  { return strcmp(p, s) < 0; }
  int operator>(char *s)  { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }
```

Операции сравнения весьма просты. Их реализация не вызывает никаких затруднений. Однако следует помнить, что левым операндом этих операций обязательно должен быть объект класса **StrType**. Если вам понадобится использовать в качестве левого операнда операции сравнения строку, заключенную в кавычки, придется добавить новые функции.

Таким образом, приведенные ниже сравнения строк вполне согласуются с определением класса.

```
StrType x("one"), y("two"), z("three");

if(x < y) cout << "x меньше y";

if(z=="three")  cout << "z равно three";

y = "o";
z = "ne";
if(x==(y+z)) cout << "x равно y+z";
```

Прочие строковые функции

В классе **StrType** определены еще три функции, предназначенные для более тесной интеграции этого класса со средой программирования на языке C++. К ним относятся функции **strsize()**, **makestr()** и функция преобразования типа **operator char* ()**. Вот как выглядят их определения.

```
int strsize() { return strlen(p); }      // Возвращает размер строки.
void makestr(char *s) { strcpy(s, p); } // Создает строку,
                                        // взятую в кавычки.
operator char *(){ return p; }           // Преобраз в тип char *.
```

Первые функции вполне очевидны. Функция **strlen()** возвращает длину строки, на которую ссылается указатель **p**. Поскольку длина этой строки может отличаться от значения, хранящегося в поле **size** (например, вследствие присваивания более короткой строки), длина строки вычисляется с помощью функции **strlen()**. Функция **makestr()** копирует строку, адресуемую указателем **p**, в символьный массив. Эта функция оказывается полезной, если необходимо превратить объект класса **StrType** в обычную строку.

Функция преобразования **operator char* ()** возвращает указатель **p**, который ссылается на строку, хранящуюся в объекте. Эта функция позволяет использовать объект класса **StrType** везде, где требуется обычная строка, завершающаяся нулевым байтом. Например, следующий фрагмент является вполне корректным.

```
StrType x("Hello");
char s[20];

// Копирование строкового объекта с помощью функции strcpy().
strcpy(s, x); // Автоматическое преобразование в тип char *.
```

Напомним, что функция преобразования выполняется автоматически, если объект является частью выражения, для которого определено это преобразование. В данном случае, поскольку прототип функции **strcpy()** сообщает компилятору, что второй аргумент имеет тип **char ***, автоматически выполняется преобразование типа **StrType** в тип **char ***, и возвращается указатель на строку, содержащуюся в объекте **x**. Затем этот указатель используется функцией **strcpy()** для копирования строки в аргумент **s**. Благодаря функции преобразования вместо аргумента типа **char *** в любой функции можно использовать объект класса **StrType**.

На заметку

*Преобразование в тип **char *** нарушает принцип инкапсуляции, поскольку, если функция получает указатель на строку, она может модифицировать ее непосредственно, без помощи функций — членов класса **StrType** и самого объекта. По этой причине преобразование в тип **char *** следует применять осторожно. Модификацию строки можно предотвратить, заставив функцию преобразования в тип **char *** возвращать константный указатель. В этом случае инкапсуляция нарушена не будет. Читатели могут сами попробовать изменить класс, руководствуясь этими указаниями.*

Полное определение класса StrType

Ниже приведена программа, содержащая полное определение класса **StrType**.

```
#include <iostream>
#include <new>
```

```
#include <cstring>
#include <cstdlib>
using namespace std;

class StrType {
  char *p;
  int size;
public:
  StrType();
  StrType(char *str);
  StrType(const StrType &o); // Конструктор копирования.
  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // Присваивание объекта
                                 // класса StrType.
  StrType operator=(char *s);    // Присваивание строки,
                                 // взятой в кавычки.
  StrType operator+(StrType &o); // Конкатенация объектов
                                 // класса StrType.
  StrType operator+(char *s);    // Конкатенация строк,
                                 // взятых в кавычки.
  friend StrType operator+(char *s, StrType &o); /*  Конкатенация
строки, взятой в кавычки, и объекта класса StrType. */

  StrType operator-(StrType &o); // Вычитание подстроки.
  StrType operator-(char *s);    // Вычитание строки,
                                 // взятой в кавычки.

  // Операторы сравнения объектов класса StrType.
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o)  { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o)  { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // Операции сравнения объектов класса StrType
  // и строк, взятых в кавычки.
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s)  { return strcmp(p, s) < 0; }
  int operator>(char *s)  { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // Возвращает размер строки.
  void makestr(char *s) { strcpy(s, p); } // Обычная строка,
                          // завершающаяся нулевым байтом.
  operator char *() { return p; } // Преобразование в тип char *.
};

// Без явной инициализации.
StrType::StrType() {
  size = 1; // Создаем место для нулевого символа.
  try {
```

```
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(p, "");
}

// Инициализация с помощью строки, взятой в кавычки.
StrType::StrType(char *str) {
  size = strlen(str) + 1; // Создаем место для нулевого символа.
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Allocation error\n";
    exit(1);
  }
  strcpy(p, str);
}

// Инициализация с помощью объекта класса StrType.
StrType::StrType(const StrType &o) {
  size = o.size;
  try {
    p = new char[size];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(p, o.p);
}

// Вывод строки.
ostream &operator<<(ostream &stream, StrType &o)
{
  stream << o.p;
  return stream;
}

// Ввод строки.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // Размер выбран произвольно -
               // при желании его можно изменить.
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    o.size = len;
```

```
  }
  strcpy(o.p, t);
  return stream;
}

// Присваивание объекту класса StrType
// другого объекта этого типа.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete [] p; // Освобождаем старую область памяти
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Присваивание строки, взятой в кавычки,
// объекту класса StrType.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Ошибка при распределении памяти\n";
      exit(1);
    }
    size = len;
  }
  strcpy(p, s);
  return *this;
}

// Конкатенация двух объектов класса StrType.
StrType StrType::operator+(StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;
  len = strlen(o.p) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
```

```
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, o.p);

  return temp;
}

// Конкатенация объекта класса StrType
// и строки, взятой в кавычки.
StrType StrType::operator+(char *s)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(temp.p, p);

  strcat(temp.p, s);

  return temp;
}

// Конкатенация строки, взятой в кавычки,
// с объктом класса StrType.
StrType operator+(char *s, StrType &o)
{
  int len;
  StrType temp;

  delete [] temp.p;

  len = strlen(s) + strlen(o.p) + 1;
  temp.size = len;
  try {
    temp.p = new char[len];
  } catch (bad_alloc xa) {
    cout << "Ошибка при распределении памяти\n";
    exit(1);
  }
  strcpy(temp.p, s);

  strcat(temp.p, o.p);

  return temp;
}
```

```
// Вычитание подстроки из объекта класса StrType.
StrType StrType::operator-(StrType &substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr.p) { // Если символ не является
                         // первой буквой подстроки,
      temp.p[i] = *s1;   // копируем его в объект temp.
      s1++;
    }
    else {
      for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++) ;
      if(!substr.p[j]) { // Если подстрока  обнаружена,
                         // удаляем ее.
        s1 += j;
        i--;
      }
      else { // Если нет, продолжаем копирование.
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}

// Вычитание строки, взятой в кавычки, из
// объекта класса StrType.
StrType StrType::operator-(char *substr)
{
  StrType temp(p);
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // Если символ не является
                       // первой буквой подстроки,
      temp.p[i] = *s1; // копируем его в объект temp.
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // Если подстрока обнаружена,
                       // удаляем ее.
        s1 += j;
        i--;
      }
      else { // Если нет, продолжаем копирование.
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
```

```
  temp.p[i] = '\0';
  return temp;
}

int main()
{
  StrType s1("Работа со строковыми объектами.\n")
  StrType s2(s1);
  StrType s3;
  char s[80];

  cout << s1 << s2;

  s3 = s1;
  cout << s1;

  s3.makestr(s);
  cout << "Преобразование в строку: " << s;

  s2 = "Это — новая строка.";
  cout << s2 << endl;

  StrType s4(" И эта тоже.");
  s1 = s2+s4;
  cout << s1 << endl;

  if(s2==s3) cout << "Строки равны.\n";
  if(s2!=s3) cout << "Строки не равны.\n";
  if(s1<s4)  cout << "Строка s1 меньше строки s4\n";
  if(s1>s4)  cout << "Строка s1 больше строки s4\n";
  if(s1<=s4) cout << "Строка s1 меньше или равна строке s4\n";
  if(s1>=s4) cout << "Строка s1 больше или равна строке s4\n";

  if(s2 > "ABC") cout << "Строка s2 больше строки ABC\n\n";

  s1 = "one two three one two three\n";
  s2 = "two";
  cout << "Исходная строка: " << s1;
  cout << "Строка после вычитания подстроки two: ";
  s3 = s1 - s2;
  cout << s3;

  cout << endl;
  s4 = "Всем привет,";
  s3 = s4 + " строки C++ весьма забавны\n";
  cout << s3;
  s3 = s3 - "Всем привет!";
  s3 = "Не правда ли, " + s3;
  cout << s3;

  s1 = s3 - "весьма ";
  cout << s1;
  s3 = s1;

  cout << "Введите строку: ";
  cin >> s1;
```

```
  cout << s1 << endl;
  cout << "Строка s1 состоит из " << s1.strsize()
       << " символов.\n";

  puts(s1); // Преобразование в тип char *.

  s1 = s2 = s3;
  cout << s1 << s2 << s3;

  s1 = s2 = s3 = "Пока ";
  cout << s1 << s2 << s3;

  return 0;
}
```

Программа выводит на экран следующие строки.

```
Работа со строковыми объектами.
Работа со строковыми объектами.
Работа со строковыми объектами.
Преобразование в строку: Работа со строковыми объектами.
Это — новая строка.
Это — новая строка. И эта тоже.
Строки не равны.
Строка s1 больше строки s4
Строка s1 больше или равна строке s4
Строка s2 больше строки ABC
Исходная строка: one two three one two three
Строка после вычитания подстроки two: one three one three

Всем привет, строки C++ весьма забавны
Не правда ли, строки C++ весьма забавны
Не правда ли, строки C++ забавны
Введите строку: I like C++
Строка s1 состоит из 10 символов
I like C++
Не правда ли, строки C++ забавны
Не правда ли, строки C++ забавны
Не правда ли, строки C++ забавны
Пока Пока Пока
```

Применение класса StrType

В завершение главы рассмотрим два примера, иллюстрирующих применение класса **StrType**. Как мы увидим, благодаря операторам, определенным в этом классе, и возможности преобразования его объектов в тип **char *** класс **StrType** полностью совместим со средой программирования на языке C++. Иначе говоря, он ничем не отличается от любого другого типа, предусмотренного в языке C++.

В первом примере создается простой словарь, состоящий из объектов класса **StrType**. Сначала он создает двухмерный массив, состоящий из объектов класса **StrType**. Первым элементом каждой пары строк является ключевое слово, а вторая строка содержит список синонимов и родственных слов. Программа предлагает пользователю ввести строку, и, если слово содержится в словаре, на экран выводится список синонимов. Эта программа весьма проста, но следует заметить, что ее ясность и прозрачность являются результатом применения класса **StrType**. (Определение класса **StrType** содержится в заголовочном файле **STR.H**.)

```
#include "str.h"
#include <iostream>
using namespace std;

StrType thesaurus[][2] = {
  "book", "volume, tome",
  "store", "merchant, shop, warehouse",
  "pistol", "gun, handgun, firearm",
  "run", "jog, trot, race",
  "think", "muse, contemplate, reflect",
  "compute", "analyze, work out, solve"
  "", ""
};

int main()
{
  StrType x;

  cout << "Введите слово: ";
  cin >> x;

  int i;
  for(i=0; thesaurus[i][0]!=""; i++)
    if(thesaurus[i][0]==x) cout << thesaurus[i][1];

  return 0;
}
```

В следующем примере с помощью объектов класса **StrType** по заданному имени проверяется, существует ли выполняемый модуль программы. Для этого достаточно ввести в командной строке имя файла без расширения. Затем программа повторно пытается найти выполняемый модуль, добавив к его имени расширение, и сообщает результат поиска. (Если файла нет, его невозможно открыть.) После проверки очередное расширение вычитается из имени файла, и вместо него подставляется новое. Ясность и простота этого примера, как и в предыдущем случае, обеспечивается классом **StrType**.

```
#include "str.h"
#include <iostream>
#include <fstream>
using namespace std;

// Расширения выполняемых файлов.
char ext[3][4] = {
  "EXE",
  "COM",
  "BAT"
};

int main(int argc, char *argv[])
{
  StrType fname;
  int i;

  if(argc!=2) {
    cout << "Командная строка: fname\n";
    return 1;
  }
```

```
  fname = argv[1];

  fname = fname + "."; // Добавляем точку.
  for(i=0; i<3; i++) {
    fname = fname + ext[i]; // Добавляем расширение.
    cout << "Проверка " << fname << " ";
    ifstream f(fname);
    if(f) {
      cout << "- Существует\n";
      f.close();
    }
    else cout << "- Не существует\n";
    fname = fname - ext[i]; // Вычитаем расширение.
  }

  return 0;
}
```

Например, если программа называется **ISEXEC**, и существует файл **TET.EXE**, командная строка **ISEXEC TEST** выводит на экран следующие строки.

```
Проверка TEST.EXE - Существует
Проверка TEST.COM - Не существует
Проверка TEST.BAT - Не существует
```

Следует заметить, что в этой программе конструктор **ifstream** использует объект класса **StrType**. Это возможно, поскольку в этом случае автоматически вызывается функция преобразования в тип **char ***. Как видим, осторожное применение возможностей языка C++ вознаграждается полной совместимостью стандартных типов и типов, разработанных пользователем.

Принципы создания и интеграции новых типов

Как демонстрирует класс **StrType**, создавать и интегрировать новые типы в среду программирования довольно просто. Для этого нужно выполнить следующие действия.

1. Перегрузить все подходящие операторы, включая операции ввода-вывода.
2. Определить все подходящие функции преобразования.
3. Предусмотреть конструкторы, позволяющие легко создавать объекты в разнообразных ситуациях.

Одним из основных преимуществ языка C++ является его расширяемость. Не следует пренебрегать такой возможностью.

Проблема

Существует одна весьма интересная проблема. Попробуйте реализовать класс **StrType**, используя стандартную библиотеку. Иначе говоря, для хранения символов следует применить не строку, а контейнер. Вместо указателей и функций примените для работы со строками итераторы и алгоритмы.

Полный справочник по

Глава 40

Синтаксический анализ выражений

Несмотря на то что стандарт языка C++ довольно обширен, некоторые темы в нем не рассмотрены. В данной главе мы изучим одну из них: *синтаксический анализ выражений*. Программы синтаксического анализа используются для вычисления алгебраических выражений, например (10–8)*3. Они довольно полезны и применяются во многих приложениях. В то же время синтаксические анализаторы окружены ореолом таинственности. По разным причинам процедуры, использующиеся в процессе синтаксического разбора, остаются достоянием избранных. Действительно многие достаточно умудренные опытом программисты пасуют перед процессом синтаксического анализа выражений.

На самом деле синтаксический анализ выражений — весьма простая процедура. Во многих отношениях она гораздо проще других задач, возникающих перед программистами. Ее простота является следствием строгих правил алгебры. В этой главе мы рассмотрим *программу рекурсивного нисходящего анализа* (recursive-descent parser) и все вспомогательные процедуры, позволяющие вычислять арифметические выражения. Мы разработаем три версии программы синтаксического анализа выражений. Первые две версии являются обычными, а третья — обобщенной. Ее можно применять к любым числовым типам. Однако перед тем как приступить к разработке синтаксического анализатора, необходимо сделать краткий обзор выражений и правил их грамматического разбора.

Выражения

Поскольку программа синтаксического разбора вычисляет алгебраические выражения, необходимо ясно представлять себе, из каких частей состоят выражения. Хотя в принципе выражения могут содержать самую разнообразную информацию, в рамках этой главы нас будут интересовать исключительно арифметические выражения. Для наших целей мы рассмотрим выражения, состоящие из следующих компонентов.

- Числа
- Операции `+`, `–`, `/`, `*`, `^`, `%` и `=`
- Скобки
- Переменные

В нашем синтаксическом анализаторе символ `^` будет означать возведение в степень (а не логическую операцию исключающего "ИЛИ", как обычно). Символ `=` обозначает оператор присваивания. Перечисленные выше компоненты объединяются в выражения в соответствии с правилами алгебры. Приведем некоторые примеры.

```
10 – 8
(100–5)*14/6
a+b–c
10^5
a=10–b
```

Установим следующие приоритеты операторов.

высший	`+ –` (унврные)
	`^`
	`* / %`
	`+ –`
низший	`=`

Операторы, имеющие одинаковый приоритет, вычисляются слева направо.

В примерах, рассмотренных по ходу изложения, все переменные обозначаются одной буквой (иначе говоря, допускается использование лишь 26 переменных, соответствующих буквам от `A` до `Z`). Строчные и прописные буквы не различаются (буквы `a` и `A` считаются одинаковыми). В первой версии анализатора все числовые переменные приводятся к типу `double`, хотя читатели легко смогут модифицировать эту программу, настроив ее на обработку любого другого числового типа. Кроме того, для упрощения логики программы в нее включены минимальные средства проверки ошибок.

Синтаксический анализ выражений: постановка задачи

На первый взгляд, синтаксический анализ выражений кажется довольно простой задачей. Однако, чтобы лучше понять ее, попробуйте-ка вычислить выражение

```
10–2*3
```

Как известно, результат этого выражения равен 4. Несмотря на то что создать программу, вычисляющую это *арифметическое* выражение, довольно легко, возникает вопрос, как создать программу, вычисляющую правильный результат *произвольного* выражения. В качестве первого варианта можно написать следующую программу.

```
a = первый операнд
while(есть операнды)
  op = оператор
  b = второй операнд
  a = a op b
}
```

Эта программа получает первый операнд, оператор и второй операнд, а затем выполняет первую операцию, получает следующий оператор и его операнды и т.д. Однако, если этот подход положить в основу всей программы, то результатом выражения 10–2*3 будет число 24 (т.е. 8*3), а не 4, поскольку эта процедура игнорирует приоритет операторов. Нельзя просто перебирать операторы и операнды слева направо, поскольку в соответствии с правилами алгебры умножение имеет более высокий приоритет, чем вычитание. Начинающие программисты часто полагают, что это ограничение легко преодолеть, и иногда, правда, в очень редких случаях, им это удается. Однако, если учесть скобки, возведение в степень, переменные, унарные операторы и тому подобное, задача становится намного сложнее.

Несмотря на то что существует несколько способов создания программы, вычисляющей выражения, мы рассмотрим лишь наиболее простой и понятный. Кстати, именно поэтому он чаще всего применяется. Этот метод называется *рекурсивным нисходящим анализом*. По мере чтения главы читатели поймут, почему он получил такое название. (Иногда при разработке синтаксических анализаторов применяются другие методы, основанные на сложных таблицах, которые должны генерироваться другими программами. Иногда такие программы называют *табличными синтаксическими анализаторами* (table-driven parser).)

Синтаксический анализ выражения

Существует много способов синтаксического анализа и вычисления выражений. В рамках рекурсивного нисходящего анализа выражения рассматриваются как *ре-*

курсивные структуры данных (recursive data structures). Если бы выражения могли состоять лишь из операций +, –, *, / и скобок, то все выражения можно было бы определить по следующим правилам.

```
выражение -> терм [+терм] [–терм]
терм -> фактор [*фактор] [/фактор]
фактор -> переменная, число или выражение
```

Квадратные скобки содержат необязательный элемент, а символ -> интерпретируется как глагол "*порождает*". Правила, указанные выше, часто называют *порождающими правилами* (production rules), или *продукциями*. Следовательно, определение терма можно было бы озвучить так: "Терм порождает фактор, умноженный на фактор, или фактор, деленный на фактор". Обратите внимание на то, что порождающие правила неявно учитывают приоритет операций.

Выражение

```
10+5*B
```

состоит из двух термов: 10 и 5*B. Второй терм состоит из двух факторов: 5 и B. Эти факторы состоят из одного числа и одной переменной.

С другой стороны, выражение

```
14*(7–C)
```

состоит из двух факторов: 14 и (7–C). Факторы состоят из одного числа и одного выражения, заключенного в скобки. Выражение, заключенное в скобки, состоит из двух термов: одного числа и одной переменной.

Этот процесс образует базис рекурсивного нисходящего анализа, представляющего собой набор взаимно рекурсивных функций, работающих по цепочке и реализующих порождающие правила. На каждом шаге анализатор выполняет операции, последовательность которых определена алгебраическими правилами. Чтобы продемонстрировать, как порождающие правила применяются для синтаксического анализа выражений, рассмотрим следующий пример.

```
9/3–(100+56)
```

Для синтаксического анализа следует выполнить следующие действия.

1. Получить первый терм 9/3.
2. Получить каждый фактор и поделить целые числа. Результат равен 3.
3. Получить второй фактор, (100+56). В этой точке начинается рекурсивный анализ второго подвыражения.
4. Получить все факторы и сложить их. Результат: 156.
5. Вернуться из рекурсивного вызова и вычесть 156 из 3. Ответ: –153.

Если это описание вас слегка напугало, не волнуйтесь. Ведь рекурсивный нисходяший анализ выражений — довольно сложная концепция. Просто не забывайте две важные вещи. Во-первых, порождающие правила неявно учитывают приоритет операторов. Во-вторых, этот метод анализа и вычисления выражений очень похож на способ, с помощью которого люди сами вычисляют математические выражения.

В оставшейся части главы мы рассмотрим три программы синтаксического анализа выражений. Первая программа анализирует и вычисляет выражения типа `double`, состоящие исключительно из констант. Затем мы рассмотрим синтаксический анализатор, позволяющий применять переменные. И в заключение третья версия анализатора будет реализована в виде шаблонного класса, который можно применять для синтаксического анализа выражений любого типа.

Класс parser

В основе программы синтаксического анализа выражений лежит класс **parser**. Первая версия этого класса приведена ниже. Последующие версии представляют собой модификации первой.

```
class parser {
  char *exp_ptr;    // Ссылается на выражение.
  char token[80];   // Хранит текущую лексему.
  char tok_type;    // Хранит тип лексемы.

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public;
  parser();
  double eval_exp(char *exp);
}
```

Класс **parser** содержит три закрытые переменные-члены. Вычисляемое выражение содержится в обычной строке, на которую ссылается указатель **exp_prt**. Таким образом, анализатор вычисляет выражения, содержащиеся в стандартных ASCII-строках. Например, анализатор способен вычислить следующие выражения:

```
"10-5"
"2 * 3.3 / 3.1416 * 3.3"
```

Если анализ начинается с выражения, указатель **exp_ptr** должен ссылаться на первый символ строки. В ходе работы анализатор считывает оставшуюся часть строки, пока не обнаружит ее конец.

Предназначение остальных переменных-членов, **token** и **tok_type**, описано в следующем разделе.

Отправной точкой анализа является функция **eval_exp()**, которой следует передать указатель на анализируемое выражение. Функции **eval_exp2()** – **eval_exp6()** вместе с функцией **atom()** образуют основу рекурсивного нисходящего анализа. Они реализуют порождающие правила, описанные выше. В последующих версиях анализатора к ним будет добавлена функция **eval_exp1()**.

Функция **serror()** предназначена для обработки синтаксических ошибок, содержащихся в выражении. Функции **get_token()** и **isdelim()** используются для разбора выражения на составные части.

Разбор выражения на составные части

Чтобы вычислить выражение, необходимо разбить его на составляющие. Поскольку эта операция является главной, ее следует рассмотреть в первую очередь.

Каждый компонент выражения называется *лексемой* (token). Например, выражение

```
A*B-(W+10)
```

состоит из лексем A, *, B, –, (, W, +, 10 и). Каждая лексема представляет собой неделимую часть выражения. Как правило, для синтаксического анализа необходима функция, которая последовательно возвращала бы отдельные лексемы, из которых состоит выражение. Кроме того, эта функция должна игнорировать пробелы и знаки табуляции, а также распознавать конец выражения. Функция, которая позволяет выделить лексемы, называется **get_token()**. Она является членом класса **parser**.

Нам необходимо знать тип лексем. В нашей программе использованы три типа лексем: **VARIABLE**, **NUMBER** и **DELIMITER**. (К типу **DELIMITER** относятся операторы и скобки.)

Функция **get_token()** показана ниже. Она извлекает следующую лексему из выражения, на которое ссылается указатель **exp_ptr**, и заносит ее в переменную **token**. Тип извлеченной лексемы записывается в переменную **tok_type**.

```
// Извлекает следующую лексему.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // В конце выражения.

  while(isspace(*exp_ptr)) ++exp_ptr; // Пропуск разделителей.

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // Переход к следующему символу.
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Если параметр c является разделителем,
// возвращает значение true.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

Присмотримся к этим функциям. После первых инициализаций функция **get_token()** проверяет, не обнаружен ли конец файла. Для этого она проверяет символ, на который ссылается указатель **exp_ptr**. Если его значение равно нулю, значит, достигнут конец выражения. Если в выражении остались еще не извлеченные лексемы, функция **get_token()** пропускает все ведущие разделители. Поскольку разделители игнорируются, указатель **exp_ptr** может ссылаться либо на число, либо на переменную, либо на оператор, либо на конец выражения (в этом случае он равен нулю). Если следующим символом является оператор, он возвращается в виде строки и записывается в переменную **token**, а в переменную **tok_type** записывается константа **DELIMITER**. Если следующий символ является буквой, предполагается, что лексема является переменной. Она возвращается в виде строки и записывается в переменную **token**, а в переменную **tok_type** записывается константа **VARIABLE**. Если следующий символ является цифрой, считывается все число. Затем оно превращается в строку и записывается в переменную **token**, а в переменную **tok_type** записывается константа **NUMBER**. В заключение, если следующий символ не относится ни к одному из перечисленных выше типов, считается, что функция **get_token()** достигла конца выражения.

Как указывалось выше, чтобы не загромождать код функции, в ней пропущены многочисленные проверки ошибок и сделано несколько упрощающих предположений. Например, предполагается, что выражение не может содержать неопознанных символов. Кроме того, в данной версии программы переменные могут иметь произвольную длину, но значимым считается только первый символ имени. Все необходимые проверки ошибок читатели могут добавить сами, ориентируясь на требования своих собственных приложений.

Чтобы лучше понять процесс разделения выражения на лексемы, рассмотрим пример.

```
A + 100 - (B*C)/2
```

Вот какие лексемы и типы возвращает функция **get_token()** в этом примере.

Лексема	*Тип лексемы*
A	**VARIABLE**
+	**DELIMITER**
100	**NUMBER**
–	**DELIMITER**
(	**DELIMITER**
B	**VARIABLE**
*	**DELIMITER**
C	**VARIABLE**
)	**DELIMITER**
/	**DELIMITER**
2	**NUMBER**
null	**null**

Напомним, что переменная **token** всегда хранит строку, завершающуюся нулевым байтом, даже если в ней записан единственный символ.

Простая программа синтаксического анализа выражений

Рассмотрим первую версию синтаксического анализатора. Он вычисляет значение выражений, состоящих лишь из констант, операторов и скобок, и не допускает выражений, содержащих переменные.

```
/* Программа, выполняющая рекурсивный нисходящий анализ
   выражений, не содержащих переменные.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

class parser {
  char *exp_ptr;  // Ссылается на выражение.
  char token[80]; // Хранит текущую лексему.
  char tok_type;  // Хранит тип лексемы.

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};

// Конструктор синтаксического анализатора.
parser::parser()
{
  exp_ptr = NULL;
}

// Отправная точка синтаксического анализа.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // Выражение пусто.
```

```
    return 0.0;
  }
  eval_exp2(result);
  if(*token) serror(0); // Последней лексемой должен быть
                        // нулевой символ.
  return result;
}

// Складываем или вычитаем два терма.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Умножаем или делим два фактора.
void parser::eval_exp3(double &result)
{
  register char op;
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Возведение в степень
void parser::eval_exp4(double &result)
{
  double temp, ex;
```

```
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Вычисление унарных операций + или -.
void parser::eval_exp5(double &result)
{
  register char  op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Распознавание скобок.
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Получает число.
void parser::atom(double &result)
{
  switch(tok_type) {
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}
```

```
// Выводит сообщение о синтаксической ошибке.
void parser::serror(int error)
{
  static char *e[]= {
       "Синтаксическая ошибка",
       "Нарушен баланс скобок",
       "Выражение пусто"
  };
  cout << e[error] << endl;
}

// Извлекает следующую лексему.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // В конце выражения.

  while(isspace(*exp_ptr)) ++exp_ptr; // Пропуск разделителя.

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // Переход на следующий символ.
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Если параметр c является разделителем,
// возвращает значение true.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

Эта программа синтаксического анализа может выполнять операции **+**, **–**, *****, **/** и **%**. Кроме того, она может возводить число в целую степень и выполнять унарную операцию **–**. Программа синтаксического анализа правильно вычисляет баланс скобок. Фактическое вычисление выражения осуществляется взаимно рекурсивными функциями **eval_exp2()**– **eval_exp6()**, а также функцией **atom()**, возвращающей извлеченное число. Предназначение каждой функции описано в комментариях.

Продемонстрируем применение этой программы.

```
int main()
{
  char expstr[80];

  cout << "Для окончания работы введите точку.\n";

  parser ob; // Создание анализатора.

  for(;;) {
    cout << "Введите выражение: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Ответ: " << ob.eval_exp(expstr) << "\n\n";
  };

  return 0;
}
```

Вот какие результаты можно получить с помощью этой программы.

```
Для окончания работы введите точку.
Введите выражение: 10-2*3
Ответ: 4

Введите выражение: (10-2)*3
Ответ: 24

Введите выражение: 10/3
Ответ: 3.33333

Введите выражение: .
```

Принципы работы синтаксического анализатора

Чтобы понять, как работает синтаксический анализатор, применим его к выражению

```
10-3*2
```

При вызове функция **eval_exp()**, представляющая собой отправную точку анализа, извлекает из выражения первую лексему. Если она является нулем, функция выводит на экран сообщение "**Выражение пусто**" и возвращает управление вызывающему модулю. Однако в данном случае лексема содержит число 10. Поскольку первая лексема не равна нулю, вызывается функция **eval_exp2()**. В результате функция **eval_exp2()** вызывает функцию **eval_exp3()**, а функция **eval_exp3()** вызывает функцию **eval_exp4()**, которая, в свою очередь, вызывает функцию **eval_exp5()**. Затем функция **eval_exp5()** проверяет, является ли лексема унарным плюсом или минусом, и, если нет, вызывает функцию **eval_exp6()**. В этой точке программы функция **eval_exp6()** либо рекурсивно вызывает функцию **eval_exp2()** (если выражение содержит скобки), либо вызывает функцию **atom()**, возвращающую число. Поскольку лексема не является открывающей скобкой, выполняется функция **atom()**, и переменной **result** присваивается число **10**. Затем из выражения извлекается следующая лексема, и функции возвращают управление по цепочке. Поскольку следующая лексема содержит знак **-**, управление передается функции **eval_exp2()**.

Затем наступает очень важный момент. Поскольку лексема содержит знак **-**, он записывается в переменную **op**. Затем анализатор извлекает следующую лексему, рав-

ную числу **3**, и вновь начинается нисходящий анализ по цепочке. Сначала, как и прежде, вызывается функция **atom()**, которая возвращает число **3**. Затем это число присваивается переменной **result**, после чего считывается лексема *. После этого управление возвращается по цепочке в функцию **eval_exp3()**, которая считывает последнюю лексему, равную числу **2**. В этот момент выполняется первая арифметическая операция — умножение **2*3**. Результат возвращается в функцию **eval_exp2()**, в которой выполняется вычитание. Результатом вычитания является число 4. Хотя, на первый взгляд, процесс анализа выглядит довольно запутанным, его проверка на других примерах показывает, что все функции работают правильно.

На основе этого синтаксического анализатора можно создать простой калькулятор. Однако прежде чем внедрять этот калькулятор в более сложные программы, например, в базы данных, необходимо научить его работать с переменными. Этому посвящен следующий раздел.

Синтаксический анализатор, работающий с переменными

Все языки программирования, многие калькуляторы и электронные таблицы используют переменные для хранения величин. Поскольку во всех этих приложениях применяется синтаксический анализ выражений, необходимо научиться обрабатывать переменные. Для этого следует усовершенствовать наш синтаксический анализатор. Как указывалось выше, для обозначения переменных используются буквы от **A** до **Z**. Переменные будут храниться в массиве, принадлежащем классу **parser**. Следовательно, в класс **parser** необходимо включить новый член.

```
double vars[NUMVARS]; // Хранит значения переменных.
```

Кроме того, придется изменить конструктор класса **parser**.

```
// Конструктор класса constructor
parser::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i] = 0.0;
}
```

Как видим, переменные инициализируются нулем.

Кроме того, нам понадобится функция, определяющая значение переменной. Поскольку переменные называются буквами от **A** до **Z**, их легко использовать в качестве индексов массива **vars**, вычитая из имен переменных ASCII-код буквы **A**. Эту операцию выполняет функция-член **find_var()**.

```
// Возвращает значение переменной.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

Функция допускает длинные имена переменных, но учитывает лишь их первый символ. Читатели могут модифицировать ее по своему усмотрению.

Кроме того, нам необходимо изменить функцию **atom()**, чтобы она могла обрабатывать не только числа, но и переменные. Вот как выглядит ее новая версия.

```
// Извлекает число или значение переменной.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}
```

С технической точки зрения этих изменений вполне достаточно, чтобы выполнить синтаксический анализ выражений, содержащих переменные. Однако у нас еще нет функции, присваивающей переменным их значения. Довольно часто эта операция выполняется за пределами синтаксического анализатора, однако мы создадим свой оператор присваивания и сделаем его частью класса **parser**. Это можно выполнить по-разному. Во-первых, можно добавить в класс **parser** функцию **eval_exp1()**. Теперь цепочка рекурсивного нисходящего анализа будет начинаться с нее. Это значит, что в начале синтаксического анализа должна вызываться функция **eval_exp1()**, а не функция **eval_exp2()**.

Рассмотрим определение функции **eval_exp1()**.

```
// Присваивание.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // Сохраняем старую лексему.
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // Вычисляем индекс переменной.
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // Возвращаем текущую лексему
      // Восстанавливаем старую лексему -
      // присваивание не выполняется.
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
```

```
    else {
      get_token(); // Извлекаем следующую часть выражения exp.
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}
```

Очевидно, что эта функция предварительно просматривает выражение, чтобы определить, действительно ли следует выполнить присваивание. Это необходимо делать потому, что имя переменной всегда предшествует оператору присваивания, но само по себе не гарантирует, что за ним обязательно следует оператор присваивания. Иначе говоря, анализатор распознает выражение A=100 как операцию присваивания и может отличить его от выражения A/10. Для этого функция **eval_exp1()** считывает из входного потока следующую лексему. Если она не содержит знака равенства, лексема возвращается во входной поток с помощью функции **putback()**, которая является частью класса **parser**.

```
// Возвращает лексему во входной поток.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}
```

После сделанных изменений, класс **parser** принимает следующий вид.

```
/* Программа, выполняющая рекурсивный нисходящий анализ
   выражений, содержащих переменные.
*/

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

class parser {
  char *exp_ptr;  // Ссылается на выражение.
  char token[80]; // Хранит текущую лексему.
  char tok_type;  // Хранит тип лексемы.
  double vars[NUMVARS]; // Хранит значения переменных.

  void eval_exp1(double &result);
  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
```

```
  void atom(double &result);
  void get_token();
  void putback();
  void serror(int error);
  double find_var(char *s);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
);

// Конструктор класса parser.
parser::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i] = 0.0;
}

// Отправная точка анализа.
double parser::eval_exp(char *exp)
{
  double result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // Выражение пусто.
    return 0.0;
  }
  eval_exp1(result);
  if(*token) serror(0); // Последняя лексема должна быть
                        // нулевым символом.
  return result;
}

// Присваивание.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // Сохраняем старую лексему.
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // Вычисляем индекс переменной.
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
```

```
      putback(); // Возвращаем текущую лексему.
      // Восстанавливаем старую лексему -
      // присваивание не выполняется.
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // Извлекаем следующую часть выражения exp.
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Складываем или вычитаем два терма.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Умножаем или делим два фактора.
void parser::eval_exp3(double &result)
{
  register char op;
  double temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
```

```
        break;
      }
    }
}

// Возведение в степень.
void parser::eval_exp4(double &result)
{
  double temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = 1.0;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * (double)ex;
  }
}

// Выполнение унарных операций + или -.
void parser::eval_exp5(double &result)
{
  register char  op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Обработка выражения, содержащего скобки.
void parser::eval_exp6(double &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Извлекает число или значение переменной.
void parser::atom(double &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
```

```
      get_token();
      return;
    case NUMBER:
      result = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Возвращает лексему во входной поток.
void parser::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}

// Выводит на экран сообщение о синтаксической ошибке.
void parser::serror(int error)
{
  static char *e[]= {
      "Синтаксическая ошибка",
      "Нарушен баланс скобок",
      "Выражение пусто"
  };
  cout << e[error] << endl;
}

// Получает следующую лексему.
void parser::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // В конце выражения.

  while(isspace(*exp_ptr)) ++exp_ptr; // Пропуск разделителя.
  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // Переход к следующему символу.
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }
```

```
  *temp = '\0';
}

// Если параметр c является разделителем,
// возвращает значение true.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Возвращает значение переменной.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

Для проверки новой версии можно вновь применить функцию **main()**, приведенную в предыдущем разделе. Теперь можно вычислять более сложные выражения, например:

```
A=10/4
A-B
C=A*(F-21)
```

Проверка синтаксических ошибок при рекурсивном нисходящем анализе

Прежде чем перейти к шаблонной версии класса **parser**, дадим краткое описание проверки синтаксических ошибок. При вычислении выражений синтаксической ошибкой считается ситуация, в которой входное выражение не соответствует строгим правилам, установленным анализатором. Как правило, это опечатки, сделанные пользователем. Например, следующие выражения будут отвергнуты анализатором.

```
10**8
(10-5)*9)
/8
```

Первое выражение содержит два оператора подряд, во втором нарушен баланс скобок, а последнее выражение начинается со знака операции. Ни одно из этих выражений анализатор не допускает. Поскольку ошибочные выражения могут привести к неверной работе программы синтаксического анализа, необходимо предусмотреть средства защиты от них.

Просматривая код класса **parser**, вы, возможно, обратили внимание на функцию **serror()**, которая вызывается в определенных ситуациях. В отличие от других методов синтаксического анализа, рекурсивный нисходящий анализ позволяет легко распознавать ошибки, поскольку они могут возникать, в основном, лишь в функциях **atom()**, **find_var()** или **eval_exp6()**, где проверяется баланс скобок. Единст-

венная проблема, связанная с распознаванием синтаксических ошибок, заключается в том, что в нашей программе не предусмотрено прекращение программы при их обнаружении. Это может привести к многократному выводу на экран повторяющихся сообщений об ошибках.

Лучше изменить функцию **serror()** так, чтобы она выполняла выход из функции. Например, все компиляторы языка C++ содержат пару функций, называемых **setjmp()** и **longjmp()**. Эти функции позволяют программе передавать управление из одной функции в *другую*. Следовательно, с помощью функции **lonjmp()** функция **setjmp()** может выполнить передачу управления в безопасную точку программы, находящуюся за пределами синтаксического анализатора.

Кроме того, в программе синтаксического анализа можно применить механизм обработки исключительных ситуаций, реализуемый с помощью операторов **try**, **catch** и **throw**.

Если оставить этот код без изменения, на экран могут выдаваться многократные сообщения о синтаксических ошибках. В одних ситуациях это может быть нежелательным, а в других желательным, поскольку так можно выявить сразу несколько ошибок. Однако, как правило, блок синтаксического контроля в коммерческих программах является более изощренным.

Создание обобщенного синтаксического анализатора

Две предыдущие версии программы предназначались для синтаксического анализа арифметических выражений типа **double**. Однако этим все многообразие выражений не исчерпывается. Кроме того, жестко заданный тип выражений ограничивает возможности синтаксического анализатора. К счастью, синтаксический анализатор можно представить в виде шаблонного класса, позволяющего обрабатывать выражения произвольных типов, как встроенных, так и пользовательских.

Рассмотрим обобщенную версию синтаксического анализатора.

```
// Обобщенный синтаксический анализатор.

#include <iostream>
#include <cstdlib>
#include <cctype>
#include <cstring>
using namespace std;

enum types { DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

template <class PType> class parser {
  char *exp_ptr;         // Ссылается на выражение.
  char token[80];        // Хранит текущую лексему.
  char tok_type;         // Хранит тип лексемы.
  PType vars[NUMVARS];   // Хранит значения переменных.

  void eval_exp1(PType &result);
  void eval_exp2(PType &result);
  void eval_exp3(PType &result);
  void eval_exp4(PType &result);
```

```
  void eval_exp5(PType &result);
  void eval_exp6(PType &result);
  void atom(PType &result);
  void get_token(), putback();
  void serror(int error);
  PType find_var(char *s);
  int isdelim(char c);
public:
  parser();
  PType eval_exp(char *exp);
};

// Конструктор класса parser.
template <class PType> parser<PType>::parser()
{
  int i;

  exp_ptr = NULL;

  for(i=0; i<NUMVARS; i++) vars[i] = (PType) 0;
}

// Отправная точка анализа.
template <class PType> PType parser<PType>::eval_exp(char *exp)
{
  PType result;

  exp_ptr = exp;

  get_token();
  if(!*token) {
    serror(2); // Выражение пусто.
    return (PType) 0;
  }
  eval_exp1(result);
  if(*token) serror(0); // Последняя лексема должна быть
                        // нулевым символом.
  return result;
}

// Присваивание.
template <class PType> void parser<PType>::eval_exp1(PType &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // Сохраняем старую лексему.
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // Вычисляем индекс переменной.
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
```

```
      putback(); // Возвращаем текущую лексему.
                 // Восстанавливаем старую лексему
                 // присваивание не выполняется.
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // Извлекаем следующую часть выражения exp.
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Складываем или вычитаем два терма.
template <class PType> void parser<PType>::eval_exp2(PType &result)
{
  register char op;
  PType temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(temp);
    switch(op) {
      case '-':
        result = result - temp;
        break;
      case '+':
        result = result + temp;
        break;
    }
  }
}

// Умножение и деление двух факторов.
template <class PType> void parser<PType>::eval_exp3(PType &result)
{
  register char op;
  PType temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
```

```
        break;
      }
    }
}

// Возведение в степень.
template <class PType> void parser<PType>::eval_exp4(PType &result)
{
  PType temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = (PType) 1;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * ex;
  }
}

// Выполнение унарных операций + или -.
template <class PType> void parser<PType>::eval_exp5(PType &result)
{
  register char  op;

  op = 0;
  if((tok_type == DELIMITER) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(result);
  if(op=='-') result = -result;
}

// Вычисляет выражение, содержащееe скобки.
template <class PType> void parser<PType>::eval_exp6(PType &result)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(result);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(result);
}

// Возвращает число или значение переменной.
template <class PType> void parser<PType>::atom(PType &result)
{
  switch(tok_type) {
    case VARIABLE:
      result = find_var(token);
```

```
      get_token();
      return;
    case NUMBER:
      result = (PType) atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

// Возвращает лексему во входной поток.
template <class PType> void parser<PType>::putback()
{
  char *t;

  t = token;
  for(; *t; t++) exp_ptr--;
}

// Выводит сообщение о синтаксической ошибке.
template <class PType> void parser<PType>::serror(int error)
{
  static char *e[]= {
      "Синтаксическая ошибка",
      "Нарушен баланс скобок",
      "Выражение пусто"
  };
  cout << e[error] << endl;
}

// Извлекает следующую лексему.
template <class PType> void parser<PType>::get_token()
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*exp_ptr) return; // В конце выражения.

  while(isspace(*exp_ptr)) ++exp_ptr; // Пропуск разделителя.

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // Переход к следующему символу.
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }
```

```
  *temp = '\0';
}

// Если параметр с является разделителем,
// возвращает значение true.
 template <class PType> int parser<PType>::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Возвращает значение переменной.
template <class PType> PType parser<PType>::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return (PType) 0;
  }
  return vars[toupper(*token)-'A'];
}
```

Тип данных, с которыми работает синтаксический анализатор, задается классом **PType**. Применение обобщенного анализатора иллюстрируется следующей программой.

```
int main()
{
  char expstr[80];

  // Демонстрация анализатора выражений типа float.
  parser<double> ob;

  cout << "Анализатор выражений типа float.  ";
  cout << "Для окончания работы введите точку \n";
  for(;;) {
    cout << "Введите выражение: ";
    cin.getline(expstr, 79);
    if(*expstr=='.') break;
    cout << "Ответ: " << ob.eval_exp(expstr) << "\n\n";
  }
  cout << endl;

  // Демонстрация анализатора целочисленных выражений.
  parser<int> Iob;

  cout << "Анализатор целочисленных выражений. ";
  cout << "Для окончания работы введите точку\n";
  for(;;) {
    cout << "Введите выражение: ";
    cin.getline(expstr, 79);
    if(*expstr==' ') break;
    cout << "Ответ: " << Iob.eval_exp(expstr) << "\n\n";
  }

  return 0;
}
```

Рассмотрим несколько примеров.

```
Анализатор выражений типа float. Для окончания работы введите точку.
Введите выражение: a=10.1
Ответ: 10.1

Введите выражение: b=3.2
Ответ: 3.2

Введите выражение: a/b
Ответ: 3.15625

Введите выражение: .

Анализатор целочисленных выражений. Для окончания работы введите
точку.
Введите выражение: a=10
Ответ: 10

Введите выражение: b=3
Ответ: 3

Введите выражение: a/b
Ответ: 3

Введите выражение: .
```

Как видим, анализатор выражений типа **float** использует действительные значения, а анализатор целочисленных выражений — целые.

Некоторые задачи

Мы уже отмечали, что наш синтаксический анализатор распознает лишь небольшое количество ошибок. Читатели могут сами усовершенствовать блок синтаксического контроля. Например, можно выделять ошибочный элемент выражения другим цветом. Это позволило бы пользователю легко обнаруживать и исправлять ошибки.

Кроме того, в нынешнем виде наш калькулятор может вычислять лишь числовые значения. Однако его можно модифицировать так, чтобы он вычислял выражения, содержащие строки, пространственные координаты или комплексные числа. Например, для вычисления строковых объектов в анализатор необходимо внести следующие изменения.

1. Определить новый тип лексемы под названием **STRING**.
2. Усовершенствовать функцию **get_token()** так, чтобы она распознавала строки.
3. Предусмотреть вариант, в котором функция **atom()** обрабатывала бы строковые лексемы.

После выполнения этих действий анализатор сможет вычислять строковые выражения, например:

```
a = "один"
b = "два"
c= a+b
```

Результат, записанный в переменной `c`, должен представлять собой конкатенацию строк `a` и `b`, т.е. быть строкой "одиндва".

Вспомним об еще одном полезном применении синтаксического анализатора: создании простого всплывающего калькулятора, вычисляющего выражения, введенные пользователем, и выводящего результат на экран. Этот калькулятор может украсить любое коммерческое приложение. Если вы программируете в среде Windows, это будет особенно просто.

Полный справочник по

Приложение А

Расширение языка C++ для платформы .NET

Технология .NET (.NET Framework), разработанная компанией Microsoft, определяет среду разработки и выполнения распределенных приложений, использующих компоненты. Она позволяет объединять разные языки программирования, обеспечивает безопасность и машинонезависимость программ, а также создает общую платформу программирования под управлением системы Windows. Несмотря на то что технология .NET появилась лишь недавно, в ближайшем будущем она, без сомнения, станет основной средой программирования на языке C++.

Технология .NET обеспечивает управляемую среду выполнения программ. Программы, предназначенные для среды .NET, не компилируются в исполняемый объектный код. Вместо этого они транслируются в промежуточный язык Microsoft Intermediate Language (MSIL), а затем выполняются под управлением среды Common Language Runtime. Управляемое выполнение является механизмом, поддерживающим основные преимущества, предлагаемые технологией .NET.

Для того чтобы воспользоваться преимуществами технологии .NET, в программе на языке C++ необходимо применять нестандартные ключевые слова и директивы препроцессора, введенные компанией Microsoft. Следует иметь в виду, что эти расширения не являются частью стандарта языка C++. Таким образом, код, использующий эти ключевые слова и директивы, не является машинонезависимым.

Описание технологии .NET и способов ее применения выходит далеко за рамки нашей книги. (Эта тема для отдельной и довольно толстой книги!) Однако мы сделаем краткий обзор расширений языка C++, связанных с этой технологией. При этом мы будем предполагать, что читатели уже знакомы с основами технологии .NET.

Дополнительные ключевые слова

Для поддержки технологии .NET компания Microsoft ввела в язык C++ новые ключевые слова.

__abstract	**__box**	**__delegate**
__event	**__finally**	**__gc**
__identifier	**__interface**	**__nogc**
__pin	**__property**	**__sealed**
__try_cast	**__typedef**	**__value**

Опишем каждое из эти слов.

Ключевое слово __abstract

Используется в сочетании с ключевым словом **__gc** для указания абстрактного управляемого класса. Класс, имеющий атрибут **__abstract**, не имеет объектов. Кроме того, такой класс не обязан содержать чисто виртуальную функцию.

Ключевое слово __box

Погружает значение в объект. Это позволяет использовать значения вместо объектов класса **System::Object**, являющегося базовым классом для всех объектов, создаваемых по технологии .NET.

Ключевое слово __delegate

Описывает представителя, инкапсулирующего указатель на функцию в управляемом классе (т.е. в классе с модификатором `__gc`).

Ключевое слово __event

Описывает функцию, представляющую событие. Задает лишь прототип этой функции.

Ключевое слово __finally

Является дополнением к стандартному механизму обработки исключительных ситуаций в языке C++. Оно используется для обозначения блока кода, который выполняется после выхода из блока `try/catch`. Блок `__finally` выполняется в любом случае.

Ключевое слово __gc

Описывает управляемый класс. Аббревиатура gc расшифровывается как "garbage collection" (уборка мусора) и означает, что объекты этого класса уничтожаются автоматически, когда становятся ненужными. Объект считается ненужным, если на него нет ни одной ссылки. Объекты класса, имеющего спецификатор `__gc`, должны создаваться с помощью оператора `new`. Массивы, указатели и интерфейсы также могут иметь спецификатор `__gc`.

Ключевое слово __identifier

Позволяет использовать ключевое слово языка C++ в качестве идентификатора. Это специальное расширение языка C++, которое очень редко используется.

Ключевое слово __interface

Описывает класс, действующий как интерфейс. В интерфейсном классе ни одна функция не имеет тела. Все функции, входящие в интерфейсный класс, неявно считаются чисто виртуальными функциями. Таким образом, интерфейс, по существу, является абстрактным классом, в котором ни одна функция не имеет реализации.

Ключевое слово __nogc

Описывает неуправляемый класс. Поскольку по умолчанию все классы считаются неуправляемыми, это ключевое слово обычно не используется.

Ключевое слово __pin

Описывает указатель, фиксирующий ячейку памяти, в которой расположен объект, на который он ссылается. Иначе говоря, "пришпиленный" (pinned) объект нельзя удалить из памяти с помощью механизма уборки мусора. В результате механизм уборки мусора не действует на указатель, имеющий модификатор `__pin`.

Ключевое слово __property

Описывает свойство, представляющее собой функцию-член, которая может извлекать и задавать значение переменной-члена. Такие функции-члены позволяют легко контролировать доступ к закрытым и защищенным членам.

Ключевое слово __sealed

Предотвращает наследование класса. С его помощью можно также предотвратить замещение виртуальной функции.

Ключевое слово __try_cast

Предусматривает попытку привести выражение к другому типу. Если приведение невозможно, генерируется исключительная ситуация типа **System::InvalidCast-Exception**.

Ключевое слово __typeof

Позволяет получить объект, инкапсулирующий информацию о заданном типе. Этот объект является экземпляром класса **System::Type**.

Ключевое слово __value

Описывает класс, представленный как тип значения. Тип значения хранит свои собственные значения. Он отличается от класса типа **__gc**, объект которого должен создаваться с помощью оператора **new**. На класс с атрибутом **__value** не распространяется действие механизма уборки мусора.

Расширение директив препроцессора

Для поддержки технологии .NET компания Microsoft предусмотрела директиву препроцессора **#using**, которая используется для импорта метаданных в программу. Метаданные содержат информацию о типе и членах, форма которой не зависит от конкретного языка программирования. Таким образом, метаданные облегчают поддержку многоязычного программирования. Все управляемые программы на языке C+ должны содержать заголовок **<mscorlib.dll>**, содержащий метаданные для платформы .NET.

Технология .NET предусматривает две прагмы. (Прагмы используются в сочетании с директивой препроцессора **#pragma**.) Первая прагма, **managed**, задает управляемый код. Вторая прагма, **unmanaged**, определяет неуправляемый (т.е. обычный) код. Эти прагмы можно использовать в программе для выбора фрагментов управляемого и неуправляемого кода.

Атрибут attribute

Компания Microsoft определила атрибут **attribute**, который используется для объявления другого атрибута.

Компилирование управляемых программ на языке C++

Пока создавалась эта книга, технологию .NET поддерживал только один компилятор — Microsoft Visual Studio .NET. Для компилирования управляемого кода следует применять опцию **/clr**, настраивающую код на систему Common Language Runtime.

Полный справочник по

Приложение Б

Язык C++ и робототехника

Долгое время я интересовался робототехникой, особенно языками управления роботами. Много лет назад я даже разработал и реализовал язык управления небольшими экспериментальными роботами. Хотя я больше не работаю профессионально в этой области, она по-прежнему очень интересует меня. С течением времени средства управления роботами сделали большой шаг вперед. Мы стоим на пороге эры робототехники. Уже существуют роботы, стригущие газоны и моющие полы. Роботы стали частью автомобилей и работают в условиях, опасных для человеческой жизни. Реальностью стали даже военные роботы. В перспективе нас ожидают еще более интересные разработки. По мере внедрения роботов в повседневную жизнь возрастает количество программистов, создающих программы управления роботами, причем большая часть этих программ написана на языке C++.

Язык C++ представляет собой естественный выбор, поскольку программы управления роботами должны быть эффективными. Особенно это относится к низкоуровневым процедурам управления двигателями, а также системам технического зрения, где необходима высокая скорость работы. Хотя некоторые части робототехнических подсистем, например, процессор распознавания естественного языка, можно написать на других языках, скажем, на C#, низкоуровневый код, лучше писать на языке C++. Таким образом, язык C++ и робототехника идут рука об руку.

Если читателей интересует робототехника, особенно создание экспериментальных роботов, им будет полезно ознакомиться с роботом, представленным на рис. Б.1. Этот робот сделал я сам. Он интересен по нескольким причинам. Во-первых, он снабжен бортовым микропроцессором, обеспечивающим выполнение основных функций управления двигателем и работу датчиков обратной связи. Во-вторых, он имеет устройство для приема и передачи информации на базе микросхемы RS-232, которое позволяет ему получать команды от главного компьютера и возвращать результаты. Благодаря этому подходу существует возможность выполнять интенсивные вычисления на удаленном компьютере, не перегружая робот дополнительными устройствами. В-третьих, он имеет видеокамеру, соединенную с беспроводным передатчиком видеосигналов.

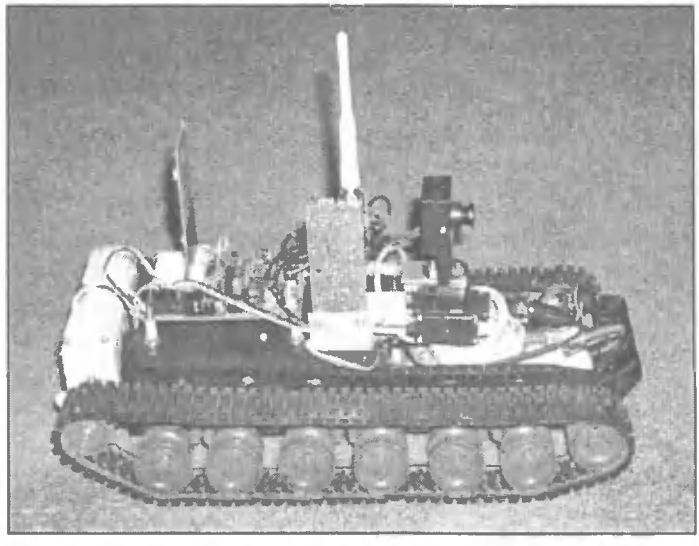


Рис. Б.1. Простой, но эффективный экспериментальный робот (фото Кена Кайзера (Ken Kaiser))

Этот робот создан на основе шасси игрушечного танка Hobbico M1 Abrams R/C. (Я обнаружил, что шасси игрушечных танков и автомобилей прекрасно подходят для создания роботов.) Я удалил все внутренние детали танка, включая приемник и средства управления скоростью, сохранив лишь моторчик. Танк Hobbico очень хорошо подходит для создания роботов, поскольку имеет крепкое шасси, хороший моторчик, может нести большой вес, и его колеса не отваливаются. Благодаря тракам он имеет нулевой радиус поворота и может перемещаться по неровной поверхности. Шасси имеет примерно 18 дюймов в длину и приблизительно 8 дюймов в ширину.

Итак, опустошив шасси, я стал добавлять к нему следующие компоненты. Для обеспечения бортового управления я использовал простой, но мощный микропроцессор BASIC Stamp 2 компании Parallax, Inc. (www.parallaxinc.com). Микросхема RS-232, видеокамера и передатчик также производятся компанией Parallax. Беспроводной передатчик RS-232 и передатчик видеосигналов действуют в радиусе 300 футов. Моторчик танка я снабдил устройством управления скоростью. Устройства такого типа применяются в скоростных гоночных автомобилях. Они управляются микропроцессором BASIC Stamp.

Вот как работает мой робот. Удаленный компьютер запускает основную программу управления роботом. Эта программа управляет основными системами его жизнедеятельности: устройствами технического зрения, средствами контроля и пространственной ориентации. Она может разучивать серию движений, а затем воспроизводить их. Удаленный компьютер передает роботу команды управления движением (через беспроводную связь на основе микросхемы RS-323). Микропроцессор BASIC Stamp получает эти команды и выполняет их. Например, получив команду "вперед", микропроцессор BASIC Stamp посылает соответствующие сигналы электронному контроллеру скорости, связанному с двигателем. Выполнив команду, робот возвращает подтверждение. Таким образом, обмен информацией между удаленным компьютером и роботом является двусторонним, и робот может подтверждать выполнение каждой команды.

Поскольку основная работа выполняется на удаленном компьютере, ее объем практически не ограничен. Например, робот может следовать за объектом, находящимся в его поле зрения. Эта способность связана с обработкой большого массива информации, которую было бы затруднительно выполнять с помощью бортового процессора.

Недавно я начал работу над механической рукой робота. Ее прототип показан на рис. Б.2. Хотя экспериментаторам и любителям робототехники доступны несколько коммерческих устройств, выполняющих функцию руки робота, я решил создать свою собственную модель, которая была бы сильнее и могла поднимать более тяжелые объекты. Механическая рука использует шаговый двигатель, вмонтированный в ее основу. Он позволяет выдвигать и вращать устройство захвата. Рука управляется своим собственным микропроцессором Stamp. Таким образом, основной контроллер робота просто передает команды управления рукой второму контроллеру. Это позволяет роботу и его механической руке выполнять параллельные операции, предотвращая перегрузку основного контроллера.

Хотя основной код управления роботом написан на языке C++, я попробовал переписать некоторые подсистемы на языке C#. Он предоставляет удобный интерфейс с устройствами идентификации положения робота и позволяет дистанционно управлять роботом через Интернет, что, как известно, является весьма трудной задачей.

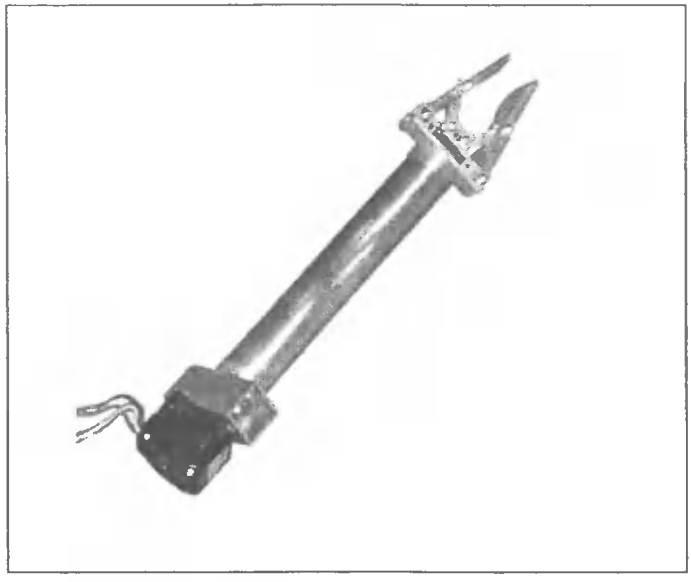


Рис. Б.2. Прототип руки робота (фото Кена Кайзера)

Предметный указатель

А

Адаптер, 506
Адресация
  косвенная, 125
  обычная, 125
Алгоритм, 505
  adjacent_difference, 718
  adjacent_find, 529; 662
  binary_search, 529; 662
  copy, 529; 662
  copy_backward, 529; 663
  count, 529; 531; 663
  count_if, 530; 531; 663
  equal, 530; 663
  equal_range, 530; 663
  fill, 530; 664
  fill_n, 530; 664
  find, 530; 664
  find_end, 530; 664
  find_first_of, 530; 664
  find_if, 530; 665
  for_each, 530; 665
  generate, 530; 665
  generate_n, 530; 665
  includes, 530; 665
  inner_product, 719
  inplace_merge, 530; 665
  iter_swap, 530; 666
  lexicographical_compare, 530; 666
  lower_bound, 530; 666
  make_heap, 530; 666
  max, 530; 667
  max_element, 530; 667
  merge, 530; 667
  min, 530; 667
  min_element, 530; 668
  mismatch, 530; 668
  next_permutation, 530; 668
  nth_element, 530; 668
  partial_sort, 530; 669
  partial_sort_copy, 530; 669
  partial_sum, 720
  partition, 530; 669
  pop_heap, 530; 669
  prev_permutation, 531; 670
  push_heap, 531; 670
  random_shuffle, 531; 670
  remove, 531; 670
  remove_copy, 531; 533; 670
  remove_copy_if, 531; 670
  remove_if, 531; 670
  replace, 531; 671
  replace_copy, 531; 533; 671
  replace_copy_if, 531; 671
  replace_if, 531; 671
  reverse, 531; 535; 672
  reverse_copy, 531; 672
  rotate, 531; 672
  rotate_copy, 531; 672
  search, 531; 672
  search_n, 531; 672
  set_difference, 531; 673
  set_intersection, 673
  set_symmetric_difference, 531; 673
  set_union, 531; 674
  sort, 531; 674
  sort_heap, 531; 674
  stable_partition, 531; 675
  stable_sort, 531; 675
  swap, 531; 675
  swap_ranges, 531; 675
  transform, 531; 535; 675
  unique, 531; 676
  unique_copy, 531; 676
  upper_bound, 531; 676
Аргумент, 46
  по умолчанию, 312

Б

Битовая маска fmtflags, 418
Битовое поле, 163
Блок, 33; 97

В

Выражение, 40; 70
  условное, 80

Д

Декрементация, 59
Деструктор, 248

Диапазон, 505
Директива
  #define, 212
  #defined, 219
  #elif, 216
  #else, 216
  #endif, 215
  #error, 214
  #if, 215
  #ifdef, 217
  #ifndef, 218
  #include, 215
  #line, 219
  #pragma, 220
  #undef, 218
Дополнительный код, 42

З

Заголовок
  algorithm, 529
  csignal, 608
  cstdio, 194
  cstdlib, 95
  cstring, 103
  exception, 411; 413; 724
  factorial, 506
  fstream, 236; 438; 625
  functional, 506
  iomanip, 625
  ios, 625
  iosfwd, 625
  iostream, 236; 625
  istream, 625
  iterator, 679
  new, 302
  numeric, 718
  ostream, 625
  sstream, 625
  stdexcept, 725
  streambuf, 625
  string, 236
  utility, 506
  vector, 236
Заголовочный файл
  stdio.h, 194
  stdlib.h, 95
  string.h, 103
Значение
  lvalue, 57
  rvalue, 57

И

Идентификатор, 42
Имя
  внешнее, 42
  внутреннее, 42
Инвертор, 506
Инициализация, 54
Инкапсуляция, 228
Инкрементация, 59
Интерфейс, 228
Исключительная ситуация
  bad_alloc, 296
  bad_cast, 466
  domain_error, 725
  invalid_argument, 725
  length_error, 725
  out_of_range, 725
  overflow_error, 725
  range_error, 725
  underflow_error, 725
  обработка, 400
Итератор, 505
  ввода, 505
  вывода, 505
  двунаправленный, 505
  обратный, 505
  произвольного доступа, 505
  прямой, 505

К

Квалификатор типа
  const, 47
  volatile, 48
Класс, 252
  accumulate, 718
  allocator, 506
  auto_ptr, 726
  back_insert_iterator, 681
  bad_exception, 413
  basic_ios, 417
  basic_iostream, 417
  basic_istream, 417
  basic_ostream, 417
  basic_streambuf, 417
  basic_string, 694
  bitset, 643
  char_traits, 701
  complex, 704
  deque, 645; 654
  exception, 413
  failure, 628

Класс
filebuf, 438
front_insert_iterator, 682
fstream, 438; 631
функция-член
close, 440
eof, 447
is_open, 440
open, 438; 633
gslice, 716
ifstream, 438; 631
insert_iterator, 680
ios
функция-член
bad, 629
clear, 629
eof, 629
exceptions, 630
fail, 630
fill, 424; 630
flags, 422; 630
flush, 630
good, 633
precision, 424; 634
rdstate, 635
setf, 419
sync_with_stdio, 638
unsetf, 420; 639
width, 423; 639
ios_base, 417
iostate, 628
istream
функция-член
gcount, 631
get, 442; 631
getline, 446; 632
ignore, 449; 633
peek, 450; 634
putback, 450; 634
read, 444; 635
readsome, 635
seekg, 450; 635
seekp, 635
setf, 636
setstate, 637
str, 637
tellg, 453; 638
write, 444
istream_iterator, 683
istreambuf_iterator, 684
istringstream, 637
istrstream, 496
iterator, 679
iterator_traits, 679
list, 646
map, 648
multimap, 650
multiset, 652
numeric_limits, 728
ofstream, 438; 631
openmode, 628
ostream
функция-член
fflush, 450
put, 634
seekp, 450
tellp, 453; 638
write, 639
ostream_iterator, 684
ostreambuf_iterator, 685
ostringstream, 637
ostrstream, 496
pair, 727
priority_queue, 655
raw_storage_iterator, 728
reverse_iterator, 682
seekdir, 628
set, 655
slice, 716
stack, 657
string, 542
stringstream, 637
strstream, 496
type_info, 458; 728
vallaray, 706
vector, 658
абстрактный, 376
базовый, 247
виртуальный, 364
вложенный, 274
дружественный, 261
локальный, 274
обобщенный, 389
производный, 247
Ключевое слово, 35
_ _abstract, 780
_ _box, 780
_ _delegate, 781
_ _event, 781
_ _finally, 781
_ _gc, 781
_ _identifier, 781
_ _interface, 781
_ _nogc, 781
_ _pin, 781
_ _property, 781

Ключевое слово
_ _sealed, 782
_ _try_cast, 782
_ _typeof, 782
_ _value, 782
auto, 44
catch, 400
class, 237
const, 47
export, 397
mutable, 488
namespace, 476
overload, 312
static, 45
template, 380
throw, 400
try, 400
typename, 397
virtual, 371
volatile, 48; 489
Комментарий, 221
многострочный, 222
однострочный, 222
Компиляция
раздельная, 37
условная, 215
Константа, 54
булева, 60
false, 60
true, 60
символьная, 54
управляющая, 56
строковая, 55
Конструктор, 247
копирования, 308
с одним параметром, 267
с параметрами, 265
явный, 489
Контейнер, 504
bitset, 506
deque, 506
list, 506; 516
map, 506; 525
multimap, 506
multiset, 506
priority_queue, 506
queue, 506
set, 506
stack, 506
vector, 506; 508
ассоциативный, 504
последовательный, 504
Курсор
записи, 451
чтения, 451
Куча, 130

Л

Литерал, 54

М

Макрос
_ _cplusplus, 221
_ _DATE_ _, 221
_ _FILE_ _, 221
_ _LINE_ _, 221
_ _STDC_ _, 221
_ _TIME_ _, 221
assert, 602
EOF, 194
EXIT_FAILURE, 95; 602
EXIT_SUCCESS, 95; 602
FOPEN_MAX, 194
MB_CUR_MAX, 602
NULL, 194; 602
RAND_MAX, 602
SEEK_CUR, 205
SEEK_END, 205
SEEK_SET, 205
SIG_DFL, 608
SIG_IGN, 608
SIGABRT, 607
SIGFPE, 607
SIGILL, 607
SIGINT, 607
SIGSEGV, 607
SIGTERM, 607
функциональный, 214
Манипулятор формата, 425
boolalpha, 425
dec, 425
endl, 425
ends, 425
fixed, 425
flush, 425
hex, 425
internal, 425
left, 425
noboolalpha, 425
noshowbase, 425
noshowpoint, 425

Манипулятор формата
noshowpos, 425
noskipws, 425
nounitbuf, 425
nouppercase, 425
oct, 425
resetiosflags, 425
right, 425
scientific, 425
setbase, 425
setfill, 425
setiosflags, 425
setprecision, 425
setw, 425
showbase, 425
showpos, 425
skipws, 425
unitbuf, 425
uppercase, 425
ws, 425
Массив, 100
безразмерный, 112
двухмерный, 104
инициализированный, 282
многомерный, 108
объектов, 280
одномерный, 100
строк, 107
указателей, 124
Метка
case, 81
default, 81
Модификатор
типа, 184
long, 41
short, 41
signed, 41
unsigned, 41
точности, 183
формата
#, 185
*, 185

H

Наследование, 229; 243
защищенное, 352
множественное, 353

O

Обобщенная функция
конкретизация, 381
специализация, 381; 382
Обобщенный класс
специализация, 389; 396
Объединение, 68; 165; 256
безымянное, 257
Объект, 228
возвращение из функции, 277
передача функции, 275
Объявление, 49
Оператор, 56
?:, 65
asm, 494
catch, 400
delete, 295
namespace, 237
new, 295
альтернатива nothrow, 302
буферизованный, 302
sizeof, 67
throw, 400
try, 400
typedef, 170
typeid, 458
using, 479
арифметический
%, 59
*, 59
. (, 59
+, 59
++, 59
взятия адреса &, 66
доступа к члену массива [], 69
доступа к члену структуры . (точка), 68
логический
!, 61
&&, 61
||, 61
перегруженный, 243
перехода
break, 82; 94
continue, 96
goto, 93
return, 93; 143
побитовый
&, 62
^, 62
|, 62
~, 62

<<, 62
>>, 62
повышения приоритета (), 69
последовательного вычисления , (запятая), 68
препроцессора
#, 220
##, 220
приведения типа
const_cast, 471
dynamic_cast, 465
reinterpret_cast, 473
static_cast, 473
присваивания, 57
сокращенный, 72
составной, 72
пустой, 97
разрешения области видимости
:, 274
разыменования указателя *, 66
составной, 97
сравнения
!=, 61
<, 61
<=, 61
>, 61
>=, 61
ссылки на член структуры ->, 68
условный
if, 75
switch, 81
вложенный if, 76
тернарная альтернатива, 78
цепочка if-then-else, 77
фиктивный, 97
цикла
do-while, 91
for, 84
while, 89
Определение, 49

П

Параметр, 46
формальный, 45
Перегрузка
конструктора, 305
оператора, 322
(), 342
, (, 344
[], 339
++, 329
->, 343
delete, 332
new, 332
бинарного, 322
унарного, 322
функции, 304
шаблонной, 384
Передача параметров
по значению, 137
по ссылке, 137; 138
Переменная, 43
автоматическая, 43
глобальная, 46
локальная, 43
статическая
глобальная, 52
локальная, 51
член класса, 238
статическая, 267
Перечисление, 167
Подкласс, 247
Полиморфизм, 228
динамический, 247
Поток, 192
бинарный, 193
ввода тбуферизованный, 497
ввода-вывода буферизованный, 499
вывода буферизованный, 496
стандартный, 208
cerr, 418
cin, 418
clog, 418
cout, 418
stderr, 208
stdin, 208
stdout, 208
текстовый, 193
Предикат
бинарный, 506
унарный, 506
Преобразование типов, 57
Препроцессор, 212
Приведение типа, 71
Признак знака, 41
Приоритет операторов, 69
Пространство имен, 236; 476
std, 483
неименованное, 480

Р

Распределение памяти динамическое, 130
Распределитель, 506
Расширение типа, 70
  целочисленное, 70
Расширенный символ, 443
Редактирование связей, 37
Редактор связей, 506; 541
  bind1st, 541
  bind2nd, 541
Рекурсия, 148

С

Связывание
  позднее, 378
  раннее, 378
Система счисления
  восьмеричная, 55
  шестнадцатеричная, 55
Спецификатор
  доступа
    private, 252
    protected, 252; 349
    public, 252
  формата
    %%, 180
    %c, 180
    %d, 180
    %e, 180
    %f, 180
    %g, 180
    %i, 180
    %n, 180
    %o, 180
    %p, 180
    %s, 180
    %u, 180
    %x, 180
  хранения, 49
    auto, 49
    extern, 49
    mutable, 49
    register, 53
    static, 51
Ссылка, 289
  возвращение из функции, 293
  на объект производного типа, 294
  независимая, 294
  передача функции, 292
Стандартная библиотека шаблонов, 504
Статус ввода-вывода, 453
Стиль, 295
Строка, 55
  завершающаяся нулевым байтом, 103
Структура, 68; 154; 254
  объявление, 154
  поле, 154
Суперкласс, 247

Т

Тип данных, 31
  bool, 60; 234
  char, 40
  double, 41
  FILE, 194
  float, 41
  fpos_t, 194
  int, 40
  long double, 41
  long int, 41
  short int, 40
  signed char, 40
  signed int, 40
  signed long int, 41
  signed short int, 41
  size_t, 194
  unsigned char, 40
  unsigned int, 40
  unsigned long int, 41
  unsigned short int, 40
  void, 41
  wchar_t, 54

У

Указатель, 66; 118
  this, 284
  на массив, 101
  на объект, 282
  на объект производного типа, 285
  на структуру, 160
  на функцию, 127
  на член класса, 287
  файла, 194
Управляющая символьная константа
  \', 56
  \", 56
  \?, 56
  \\, 56
  \0, 56

\a, 56
\b, 56
\f, 56
\n, 56
\r, 56
\t, 56
\v, 56
\xN, 56

Ф

Файл, 193
режим, 195
ios::app, 438
ios::ate, 438
ios::binary, 438
ios::in, 438
ios::out, 438
ios::trunc, 438
Флаг формата, 418
boolalpha, 419
dec, 419
fixed, 419
hex, 419
left, 419
oct, 419
right, 419
scientific, 419
showbase, 419
showpoint, 419
showpos, 419
skipws, 419
unitbuf, 419
uppercase, 419
Функтор, 506
бинарный, 537
divides, 537
equal_to, 537
greater, 537
greater_equal, 537
less, 537
less_equal, 537
logical_and, 537
logical_or, 537
minus, 537
modulus, 537
multiplies, 537
not_equal_to, 537
plus, 537
унарный, 537
logival_not, 537
negate, 537
Функция, 33; 136
abort, 401; 602
abs, 602
acos, 584
asctime, 592
asin, 584
atan, 585
atan2, 585
atexit, 603
atof, 603
atoi, 603
atol, 604
bsearch, 604
calloc, 598
ceil, 585
cfseek, 561
clearerr, 556
clock, 592
cos, 585
cosh, 585
ctime, 593
difftime, 593
div, 604
exit, 95; 401; 605
exp, 586
fabs, 586
fclose, 194; 196; 556
feof, 194; 199; 557
ferror, 194; 201; 557
fflush, 194; 203; 557
fgetc, 194; 557
fgetpos, 558
fgets, 194; 200; 558
floor, 586
fmod, 586
fopen, 194; 195; 558
fprintf, 194; 207; 559
fputc, 194; 560
fputs, 194; 200; 560
fread, 204; 560
free, 130; 598
freopen, 209; 560
frexp, 586
fscanf, 194; 207; 561
fseek, 194; 205
fsetpos, 562
ftell, 194; 562
fwrite, 204; 562
getc, 194; 197; 562
getch, 178
getchar, 175; 178; 563

getche, 178
getenv, 605
gets, 177; 178; 563
gmtime, 593
isalnum, 574
isalpha, 574
iscntrl, 574
isdigit, 575
isgraph, 575
islower, 575
isprint, 575
ispunct, 575
isspace, 575
isupper, 576
isxdigit, 576
labs, 605
ldexp, 587
ldiv, 605
localeconv, 593
localetime, 594
log, 587
log10, 587
longjmp, 606
main, 141
    аргументы argc и argv, 141
malloc, 130; 598
mblen, 606
mbstowcs, 606
mbtowc, 607
memchr, 576
memcmp, 576
memcpy, 576
memmove, 577
memset, 577
mktime, 595
modf, 587
perror, 563
pow, 587
printf, 179; 564
putc, 194; 196; 566
putchar, 175; 178; 566
puts, 178; 566
qsort, 607
raise, 607
rand, 608
realloc, 599
remove, 194; 567
remove, 203
rename, 567
rewind, 194; 200; 567
scanf, 185; 567
set_terminate, 412
set_unexpected, 412
setbuf, 570
setjmp, 608
setlocale, 595
setvbuf, 570
signal, 608
sin, 588
sinh, 588
sprintf, 570
sqrt, 588
srand, 609
sscanf, 571
strcat, 103; 577
strchr, 103; 577
strcmp, 103; 577
strcoll, 578
strcpy, 103; 578
strcspn, 578
strerror, 578
strftime, 595
strlen, 103; 579
strncat, 579
strncmp, 579
strncpy, 579
strpbrk, 580
strrchr, 580
strspn, 580
strstr, 103; 580
strtod, 609
strtok, 580
strtol, 609
strtoul, 610
strxfrm, 581
system, 610
tan, 588
tanh, 589
terminate, 411
time, 596
tmpfile, 571
tmpnam, 571
tolower, 581
toupper, 581
uncaught_exception, 413
unexpected, 411
ungetc, 571
va_arg, 611
va_end, 611
va_start, 611
vfprintf, 572
vprintf, 572

vsprintf, 572
wcstombs, 611
wctomb, 611
виртуальная, 368
возвращаемое значение, 145
дружественная, 258
область видимости, 136
обобщенная, 380
общий вид, 136
операторная, 322
  член класса, 322
перегруженная, 240
подставляемая, 262
преобразования типа, 484
прототип, 149
рекурсивная, 148
  формальные параметры, 137
  чисто виртуальная, 374
  член класса, 238
    статическая, 271

Э

Экземпляр класса, 238
Эскейп-последовательность, 56

Я

Язык
  блочно-структурированный, 32
  структурированный, 32

Научно-популярное издание

Герберт Шилдт

Полный справочник по C++, 4-е издание

Литературный редактор	*О.Ю. Белозовская*
Верстка	*В.И. Бордюк*
Художественный редактор	*В.Г. Павлютин*
Корректоры	*Л.А. Гордиенко, Л.В. Коровкина, О.В. Мишутина, Л.В. Чернокозинская*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 12.08.2005. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 64,5. Уч.-изд. л. 40.
Доп. тираж 3000 экз. Заказ № 2593.

Отпечатано с фотоформ
в ФГУП "Печатный двор" им. А. М. Горького
Федерального агентства по печати
и массовым коммуникациям.
197110, Санкт-Петербург, Чкаловский пр., 15.

ПОЛНЫЙ СПРАВОЧНИК по C,
4-Е ИЗДАНИЕ

Герберт Шилдт

www.williamspublishing.com

в продаже

В данной книге, задуманной как справочник для всех программистов, работающих на языке C, независимо от их уровня подготовки, подробно описаны все аспекты языка C и его библиотеки стандартных функций. Главный акцент сделан на стандарте ANSI/ISO языка C. Приведено описание как стандарта C89, так и C99. В книге особое внимание уделяется учету характеристик трансляторов, среды программирования и операционных систем, использующихся в настоящее время. Уже в самом начале подробно представлены все средства языка C, такие как ключевые слова, инструкции препроцессора и другие. Вначале описывается главным образом C89, а затем приводится подробное описание новых возможностей языка, введенных стандартом C99. Такая последовательность изложения позволяет облегчить практическое программирование на языке C, так как в настоящее время именно эта версия для большинства программистов представляется как "собственно C", к тому же это самый распространенный в мире язык программирования. Кроме того, эта последовательность изложения облегчает освоение C++, который является надмножеством C89. В описании библиотеки стандартных функций C приведены как функции стандарта C89, так и C99, причем функции, введенные стандартом C99, отмечены специально. Все книги Шилдта программисты всегда горячо любили за наличие содержательных, нетривиальных примеров. В книге рассматриваются наиболее важные и распространенные алгоритмы и приложения, необходимые для каждого программиста, а также применение методов искусственного интеллекта и программирование для Windows 2000.

C++: базовый курс, третье издание

Герберт Шилдт

ISBN 5-8459-0768-3
в продаже

В этой книге описаны все основные средства языка C++: от элементарных понятий до супервозможностей. После рассмотрения основ программирования на C++ (переменных, операторов, инструкций управления, функций, классов и объектов) читатель освоит такие более сложные средства языка, как механизм обработки исключительных ситуаций (исключений), шаблоны, пространства имен, динамическая идентификация типов, стандартная библиотека шаблонов (STL), а также познакомится с расширенным набором ключевых слов, используемым в программировании для .NET. Автор справочника — общепризнанный авторитет в области программирования на языках C и C++, Java и C# — включил в текст своей книги и советы программистам, которые позволят повысить эффективность их работы.

C++: РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ 2-е издание

Герберт Шилдт

www.williamspublishing.com

В этой книге описаны основные средства языка C++, которые необходимо освоить начинающему программисту. После рассмотрения элементарных понятий (переменных, операторов, инструкций управления, функций, классов и объектов) читатель легко перейдет к изучению таких более сложных тем, как перегрузка операторов, механизм обработки исключительных ситуаций (исключений), наследование, полиморфизм, виртуальные функции, средства ввода-вывода и шаблоны. Автор справочника — общепризнанный авторитет в области программирования на языках C и C++, Java и C# — включил в свою книгу множество тестов для самоконтроля, которые позволяют быстро проверить степень освоения материала, а также разделы "вопросов и ответов", способствующие более глубокому изучения основ программирования даже на начальном этапе.

ISBN 5-8459-0840-X

в продаже

Четвертое издание

Полный справочник по C++

Обновленный и дополненный классический справочник Шилдта по C++

Наиболее полный путеводитель по C++

Авторитетный профессионал и блестящий автор книг по программированию, Герберт Шилдт переработал и дополнил свой, ставший уже классическим, справочник по C++. Шилдт продемонстрировал и подобно разъяснил каждый аспект языка C++, включая ключевые слова, операторы, директивы препроцессора и библиотеки. В справочнике перечислены даже ключевые слова, используемые в программировании на платформе .NET. Все это описано кратко и доходчиво, в присущем автору стиле, принесшем ему заслуженную славу. Информация, содержащаяся в справочнике, будет полезна как начинающему программисту, так и опытному профессионалу.

В книге рассмотрены:

- типы данных и операторы
- управляющие конструкции
- функции
- классы и объекты
- конструкторы и деструкторы
- перегрузка функций и операторов
- наследование
- виртуальные функции
- пространства имен
- шаблоны
- обработка исключительных ситуаций
- библиотеки средств ввода-вывода
- стандартная библиотека шаблонов (STL)
- контейнеры, алгоритмы и итераторы
- принципы объектно-ориентированного программирования (ООП)
- динамическая идентификация типа (RTTI)
- препроцессор
- и многое другое

Герберт Шилдт — автор наиболее популярных книг по программированию. Он является признанным специалистом по языкам C, C++, Java и C#. Шилдт был членом комитета ANSI/ISO по стандартизации языка C++. Книги, написанные Шилдтом, переведены на все основные языки мира. Общий объем их продаж превышает 3 миллиона экземпляров.

Категория: программирование/язык C++
Уровень: от начинающих до опытных программистов

The McGraw-Hill Companies

Издательский дом "Вильямс"
www.williamspublishing.com

OSBORNE
www.osborne.com

Osborne delivers results!

ISBN 5-8459-0489-7